FINAL

Unit Cell Geometry of Multiaxial Preforms for Structural Composites

Final Report

Submitted to

NASA Langley Research Center

Program Director

Frank Ko

Project Engineers

Dr. Charles Lei
Dr. Anisur Rahman
Dr. G. W. Du
Mr. Yun-Jia Cai

Fibrous Materials Resaerch Center
and
Department of Materials Engineering
Drexel University

November 1993

DREXEL
UNIVERSITY

Table of Contents

Executive Summary

1. Introduction

2. Processing model
    2.1 *Background*
    2.2 *3-D braidiing*
    2.3 *Multiaxial Warp Knit*
    2.4 *References*

3. Geometric Modeling
    3.1 *Theoretical Background*
        3.1.1 Yarn Path Modeling
        3.1.2 Solid Modeling of the Yarn
    3.2 *Application to theBraid Model*
    3.3 *Application to Multiaxial Warp Knit*
    3.4 *References*

4. Visulization Verification
    4.1 *Material System*
        4.1.1 3-D Braid
        4.1.2 Multiaxial Warp Knit
        4.1.3 Sample Preparation
    4.2 *3-D braid*
        4.2.1 Computer Generated Graphs
        4.2.2 Experimental verification
    4.3 *Multiaxial Warp Knit*
        4.3.1 Computer Generated Graphs
        4.3.2 Experimental verification

5. Unit Cell Modeling
    5.1 *Background*
    5.2 *3-D Braiding*
    5.3 *Multiaxial Warp Knit*
    5.4 *References*

6. Applications
    6.1 Implementation of geometric design
    6.2 Engineering design

Appendix
    A. Programs
    B. Published papers

Executive Summary

The objective of this study is to investigate the yarn geometry of multiaxial preforms. The importance of multiaxial preforms for structural composites is well recognized by the industry but to exploit their full potential, engineering design rules must be established. This study is a step in that direction. In this work the preform geometry for knitted and braided preforms were studied by making a range of well designed samples and studying them by photo microscopy . The structural geometry of the preforms is related to the processing parameters. Based on solid modeling and B-spline methodology a software package is developed . This computer code enables real time structural representations of complex fiber architecture based on the rule of preform manufacturing. The code has the capability of zooming and section plotting. These capabilities provide a powerful means to study the effect of processing variables on the preform geometry. The code also can be extended to an auto mesh generator for downstream structural analysis using finite element method. This report is organized into six section. In the first section the scope and background of this work is elaborated. In Section two the unit cell geometries of braided and multi-axial warp knitted preforms is discussed. The theoretical frame work of yarn path modeling and solid modeling is presented in Section three. The thin section microscopy carried out to observe the structural geometry of the preforms is the subject in section four. The structural geometry is related to the processing parameters in section five. Section six documents the implementation of the modeling techniques into the computer code MP-CAD. A user manual for the software is also presented here. The source codes and published papers are listed in the Appendices.

Chapter – 1. Introduction

The important role of preforming in the chain of composite manufacturing processes has been well recognized by the composite industry in the recent years. It has been demonstrated in many cases that the mechanical properties of a composite can be improved by the pre-orientation, pre-shaping and pre-placement of matrices (as in the case of commingled thermoplastic composites). The reduction in processing steps due to preforming also contributes to the reduction of manufacturing cost of composites. The recent rise in popularity of composite production by resin transfer molding (RTM) further expand the need for advanced preforming technology. Of the large family of textile preforms, the class of multiaxial preforms presents perhaps the most promising solution to the economic manufacturing of high damage tolerant structural composites.

Considering the importance of multiaxial preforms for structural composites, it was recognized by NASA and Drexel that engineering design rules must be established in order to exploit preforms to their full potential for composites. In order for them to be useful, these design rules must be capable of linking preform processing parameters to the structural engineering design environment.

Central to this linkage is the quantification of the structural geometry or the fiber architecture of the preforms in terms of processing parameters. This geometric model provides a means for the incorporation of materials properties into a preprocessor for down stream finite element structural analysis. The fiber geometry for knitted and braided preforms were studied by making a range of well designed samples and studying them by photo microscopy.

Based on solid modeling and B-spline methodology a software package was developed at the Fibrous Materials Research Center. This computer code enables real time structural representations of complex fiber architecture based on the rule of preform manufacturing. The code has the capability of zooming and section plotting. These capabilities provide a powerful means to study the effect of processing variables on the preform geometry. The code also can be extended to an auto mesh generator for structural analysis.

Chapter 2. - Processing Model

2.1 Background

The current trend in the composite materials industry is to expand the use of composites from secondary non-load bearing to primary load bearing structural application. This requires a significant improvement of the damage tolerance and reliability of composites. In addition, it is also desirable to reduce the cost and broaden the usage of composites from aerospace to automotive and building construction applications. This calls for the development of a capability for quantity production and the net or near-net shape reinforcement of structural composites.

In order to improve the damage tolerance and delamination property of composites, a high level of through-the-thickness strength is required. The reliability of a composites depends on the uniform distribution of the materials and consistency of interfacial properties. The structural integrity, handleability and formability of the reinforcing material for the composite are critical for large scale automated production of structural shapes.

As introduced by Ko [2.1], there is a large family of textile preforms available for advanced composites ranging from 3-D integrated net-shape structures to thin and medium thickness multilayer fabrics. In addition to properties translation efficiency, structural integrity and formability, a key requirement for textile preforms for building construction is their availability at a reasonable fabrication cost. There are two families of preforms which have the potential to meet the demand for structural composites 3-D braid and multiaxial war knit (MWK) fabrics.

In order to fully understand the processes associated with three dimensional fiber network, it is necessary to develop a mathematical model of the fiber network. The fiber networks under consideration have no value to the realm of engineering unless they can be manufactured. In this section the methods of forming 3-D braids and multiaxial warp knits (MWK) using existing mechanisms are investigated. The models developed herein relate the nature of the fiber networks to the geometric properties of the resulting fabric through an understanding of the fabrication machine.

The interest of this section is to develop processing models of 3-D braid and MWK which predict the geometric parameters of the braided and knitted fabric and relate them to the

processing variables. In general, the specific geometry of a fabric to be formed is affected by the size of the yarns and their placement within the machine, and the formation of geometrical shapes can be achieved by the proper positioning of longitudinal, transverse and through thickness laid-in yarn systems. In addition to the geometrical effects of these systems, the packing of yarns within the structure will also be affected. Furthermore the mechanical properties of the resulting structures are affected.

2.2 3-D braiding

2.2.1 3-D Braiding Process

3-D braiding technology is an extension of well established two-dimensional (2-D) braiding technology in which fabric is constructed by the intertwining or orthogonal interlacing of two sets of yarns (braiders and axials) in order to form an integral structure. 3-D braids, widely used to reinforce structural composites, provide enhanced properties and the possibility for near-net-shape reinforcement, as opposed to conventional weaves which are layered to form composites.

A generalized schematic of a 3-D braiding process is shown in Figure 2.1. Axial yarns, if present in a particular braid, are fed directly into the structure from packages located below the track plate. Braiding yarns are fed from bobbins mounted on carriers that move on the track plate. The pattern produced by the motion of the braiders relative to each other and the axial yarns establish the type of braid being formed, as well as the microstructure.

3-D braids have been produced on traditional horn-gear machines for ropes and packings in solid, circular, or square cross-sections. A number of new machines without the traditional horn-gears have been developed to create 3-D braids with complex shapes. Track-and-column and 2-step braiding processes are two examples of the new developments. The mechanism of these braiding methods differs from the traditional horn gear method only in the way the carriers are displaced to create the final braid geometry. Instead of moving in a continuous Maypole fashion, as does in the solid braider, these 3-D braiding methods invariably move the carriers in a sequential, discrete manner.

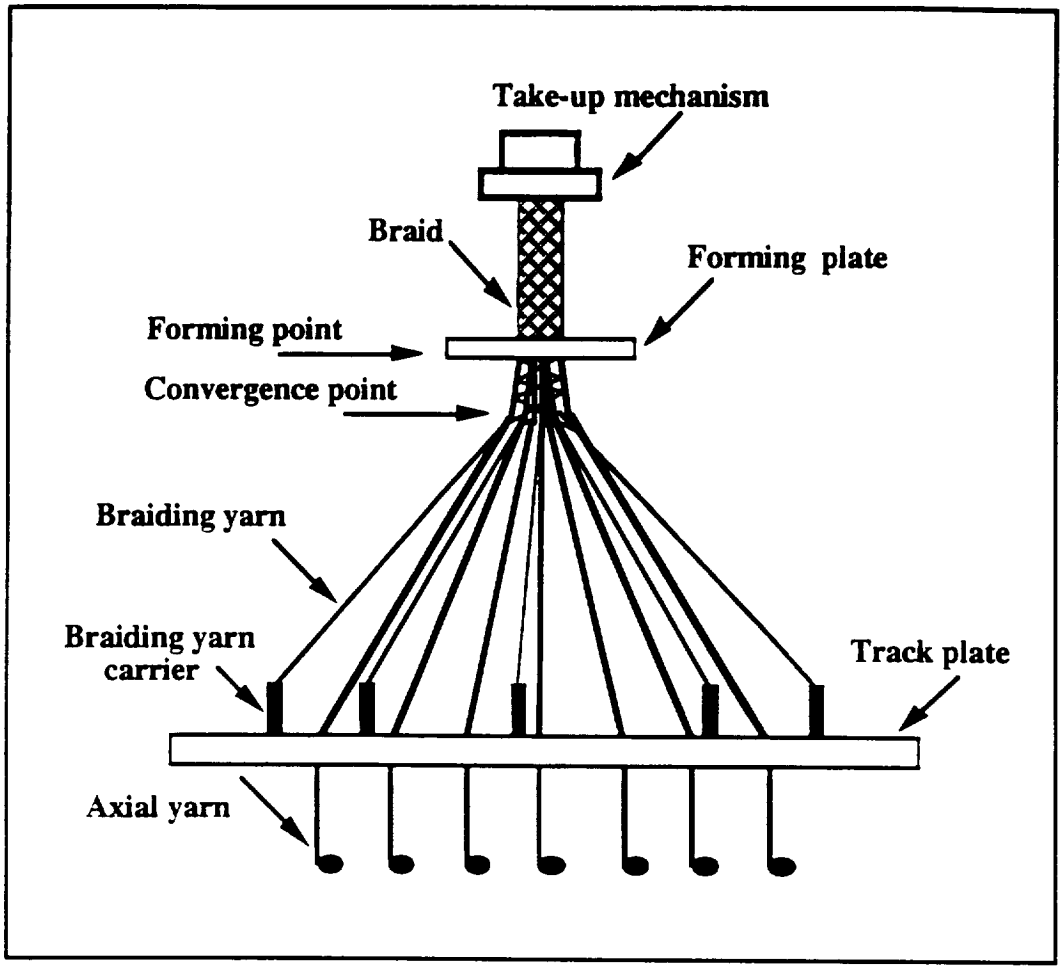


Figure 2.1 Schematic drawing of a generalized 3-D braider.

Figure 2.2 shows a basic loom setups and the carrier motions during four step braiding in a rectangular configuration. The carriers are arranged in tracks and columns to form the required shape and additional carriers are added to the outside of the array in alternating locations. Four steps of motion are imposed to the tracks and columns during a complete braiding machine cycle, resulting in the alternate X and Y displacement of yarn carriers. Since the track and column both move one carrier displacement in each step, the braiding pattern is referred to as 1x1. 0° axial reinforcements can also be added to the track-and-column braid as desired. The formation of shapes, such as T-beam and I-beam, is accomplished by the proper positioning of the carriers and the joining of various rectangular groups through selected carrier movements.

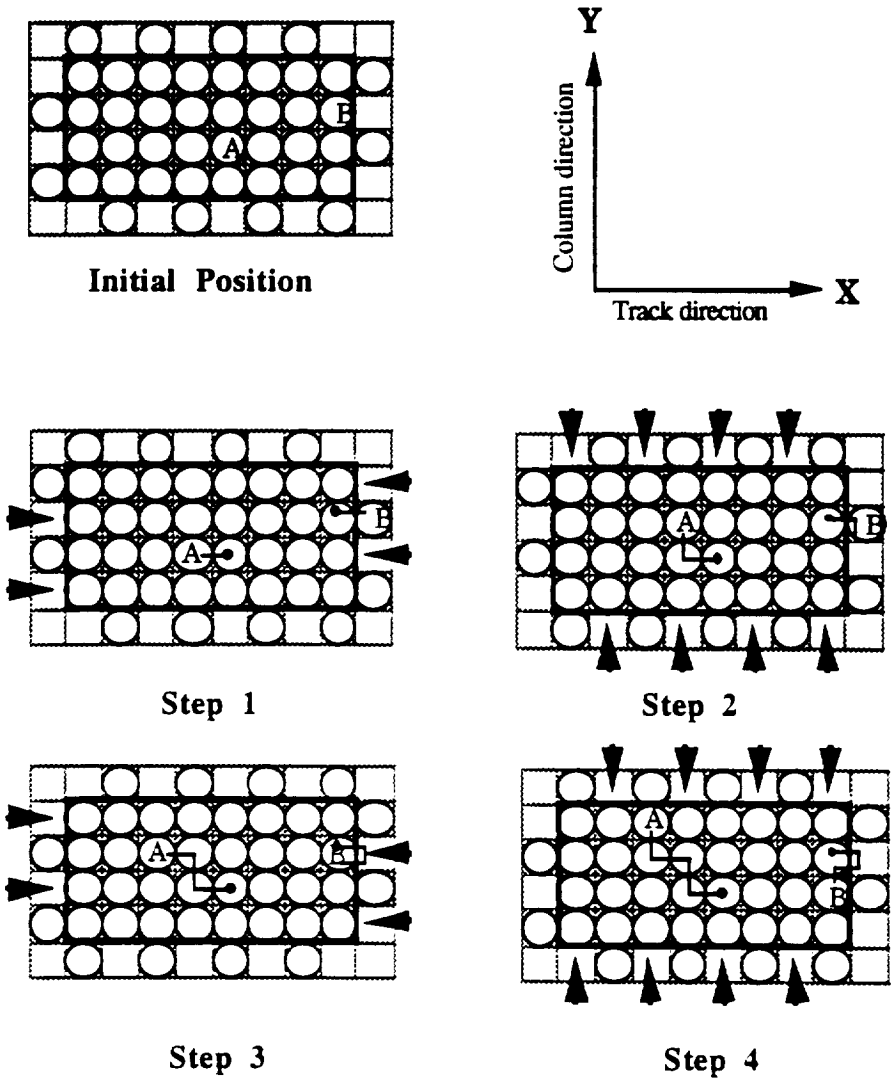


Figure 2.2 Formation of a rectangular 3-D track and column braid

2.2.2 Quantification of the Braiding Operation

The understanding and quantification of the braiding process is necessary for later identification of the unit cell geometry. One way of looking at braiding is to consider the process as a permutation operator, in which the operator maps from a two-dimensional lattice to another two-dimensional lattice and the braided fabric is considered as the trace of the map. This approach is taken by Ko and Pastore [2.2]. Although it provides a mathematical description for braiding motions, considerable simplification has to be made to apply the general theory to practice.

In the present work a direct approach is taken where the motion of yarn carriers are traced throughout a braiding cycle. This trace along with other relevant processing information is

used to generate the path of the yarn and the shape of the braided fabric. This approach can easily be translated to computer codes to simulate braiding [Chapter 6].

2.2.3 Notations and Conventions used in the Model

Geometry

The loom is consists of m tracks and n colums. The outermost tracks and columns do not move. A loom matrix B_{ij} is defined such that $B_{ij}=1$ if a carrier position contains a bobbin. Otherwise $B_{ij}=0$. The lower right corner bobbin is taken as the reference bobbin and is defined to be at position 1,1.

Braiding cycle

The four step braiding cycle, as shown in Figure 2.2 can be broken down into the following operations:

Step 1: The odd tracks move left and even tracks move right. (left -1; right +1)

Step 2: The odd columns move down and even columns move up.(up +1; down -1)

Step 3: The odd tracks move right and even tracks move left.

Step 4: The odd columns move up and even columns move down.

The model is developed for 1×1 braiding pattern, i.e, all tracks and columns only move zero- or one-carrier distance in any steps, but it can easily be extended to 1x2, 1x3, 2x3 or even complicated mixed patterns.

Relational Operators

The relational operators used in the formulation may not be apparent to all the readers and hence are explained here

Relation "→"

This is a conditional operator and is used to express an "if-then" relationship. p→q implies that if p is true q takes place.

i.e. (b>10)→a=2, implies that if b is less than 10 then a is equal to 2.

Conjunction "^"

The conjunction operator is equivalent to logical AND used in most high level programming languages.

i.e. (b>10)^(c>9)→a=2, implies if b is less than 10 AND c is less than 9 then a is equal to 2

The Position Tensor, $P_{ijk}^{p,s}$

The positon tensor traces the bobbin as the braiding progresses.

The subscripts i and j denote the bobbin that is being traced. (i,j) is the position of the bobbin at the start of braiding.

The third subscript k can have two values, T or C. T is for track and C is for column.

The superscripts p and s stand for period and steps respectively. In four step braiding each period consists of four steps. Writing out the third subscript:

$$P_{ijk}^{p,s} = \begin{Bmatrix} P_{ijT}^{p,s} \\ P_{ijC}^{p,s} \end{Bmatrix} \tag{2-1}$$

Here, $P_{ijT}^{p,s}$ is the track position of the bobbin i,j at the end of p period and s steps. Similarly, $P_{ijC}^{p,s}$ is the column position. If the position tensor is known as the braiding progresses the trace of the yarn can be easily determined.

2.2.4 Representation of the Braiding Cycle

Startup (0 cycle, 0 step)

At the startup phase all the bobbins are at their inital positons. This is represented as follows:

$$\begin{Bmatrix} P_{ijT}^{0,0} \\ P_{ijC}^{0,0} \end{Bmatrix} = \begin{Bmatrix} i \\ j \end{Bmatrix} \tag{2-2}$$

The inital cycle

Step 1: In step 1, the odd tracks move left (−1) and the even tracks move right (+1). This is formally represented as:

$$\begin{Bmatrix} P_{ijT}^{0,1} \\ P_{ijC}^{0,1} \end{Bmatrix} = \begin{Bmatrix} \left(B_{ij}=1\right) \wedge (1 < P_{ijC}^{0,0} < m) \rightarrow P_{ijT}^{0,0} + (-1)^{P_{ijC}^{0,0}} \\ P_{ijC}^{0,0} \end{Bmatrix} \tag{2-3}$$

The condition $B_{ij}=1$ in the conjunction states that we trace only the looms which have bobbins. Since this is always true, it will be omitted but implied. With this simplification:

$$\begin{Bmatrix} P_{ijT}^{0,1} \\ P_{ijC}^{0,1} \end{Bmatrix} = \begin{Bmatrix} (1 < P_{ijC}^{0,0} < m) \rightarrow P_{ijT}^{0,0} + (-1)^{P_{ijC}^{0,0}} \\ P_{ijC}^{0,0} \end{Bmatrix} \tag{2-4}$$

Step 2: In step 2, the odd columns move down (− 1) and even columns move up (+1).

$$\begin{Bmatrix} P_{ijT}^{0,2} \\ P_{ijC}^{0,2} \end{Bmatrix} = \begin{Bmatrix} P_{ijT}^{0,1} \\ (1 < P_{ijT}^{0,1} < n) \rightarrow P_{ijC}^{0,1} + (-1)^{P_{ijT}^{0,1}} \end{Bmatrix} \tag{2-5}$$

Step 3: In step 3, the odd tracks move right and the even tracks move left.

$$\begin{Bmatrix} P_{ijT}^{0,3} \\ P_{ijC}^{0,3} \end{Bmatrix} = \begin{Bmatrix} (1 < P_{ijC}^{0,2} < m) \rightarrow P_{ijT}^{0,2} - (-1)^{P_{ijC}^{0,2}} \\ P_{ijC}^{0,2} \end{Bmatrix} \tag{2-6}$$

Step 4: In step 4, the odd columns move up and even columns move down.

$$\begin{Bmatrix} P_{ijT}^{0,4} \\ P_{ijC}^{0,4} \end{Bmatrix} = \begin{Bmatrix} P_{ijT}^{0,3} \\ (1 < P_{ijT}^{0,3} < n) \rightarrow P_{ijC}^{0,3} - (-1)^{P_{ijT}^{0,3}} \end{Bmatrix} \tag{2-7}$$

Generalization to subsequent steps

Once the motion of the loom for the first braiding cycle (which consists of 4 braiding steps) is simulated, the subsequent steps can be progressively generated by equeations (2-8) to (2-11)

Step 1

$$\begin{Bmatrix} P_{ijT}^{p,1} \\ P_{ijC}^{p,1} \end{Bmatrix} = \begin{Bmatrix} (1 < P_{ijC}^{p-1,4} < m) \rightarrow P_{ijT}^{p-1,4} + (-1)^{P_{ijC}^{p-1,4}} \\ P_{ijC}^{p-1,4} \end{Bmatrix} \tag{2-8}$$

Step 2

$$\begin{Bmatrix} P_{ijT}^{p,2} \\ P_{ijC}^{p,2} \end{Bmatrix} = \begin{Bmatrix} P_{ijT}^{p,1} \\ (1 < P_{ijT}^{p,1} < n) \rightarrow P_{ijC}^{p,1} + (-1)^{P_{ijT}^{p,1}} \end{Bmatrix} \tag{2-9}$$

$$\begin{Bmatrix} P_{ijT}^{p,3} \\ P_{ijC}^{p,3} \end{Bmatrix} = \begin{Bmatrix} (1 < P_{ijC}^{p,2} < m) \rightarrow P_{ijT}^{p,2} - (-1)^{P_{ijC}^{p,2}} \\ P_{ijC}^{p,2} \end{Bmatrix} \tag{2-10}$$

$$\begin{Bmatrix} P_{ijT}^{p,4} \\ P_{ijC}^{p,4} \end{Bmatrix} = \begin{Bmatrix} P_{ijT}^{p,3} \\ (1 < P_{ijT}^{p,3} < n) \rightarrow P_{ijC}^{p,3} - (-1)^{P_{ijT}^{p,3}} \end{Bmatrix} \tag{2-11}$$

An example

We use the formulation to trace the orbit of bobbin initially at (3,3) on a (5×14) loom. Here i=j=3, m=5 and n=14.

The machine setup and the orbit of the bobbin is shown in Figure 2.3.

Initialization

$$\begin{Bmatrix} P_{33T}^{0,0} \\ P_{33C}^{0,0} \end{Bmatrix} = \begin{Bmatrix} 3 \\ 3 \end{Bmatrix} \tag{2-12}$$

inital cycle

Step 1

$$\begin{Bmatrix} P_{33T}^{0,1} \\ P_{33C}^{0,1} \end{Bmatrix} = \begin{Bmatrix} 3+(-1)^3 \\ 3 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 3 \end{Bmatrix} \tag{2-13}$$

Step 2

$$\begin{Bmatrix} P_{33T}^{0,2} \\ P_{33C}^{0,2} \end{Bmatrix} = \begin{Bmatrix} 2 \\ 3+(-1)^2 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4 \end{Bmatrix} \tag{2-14}$$

Step 3

$$\begin{Bmatrix} P_{33T}^{0,3} \\ P_{33C}^{0,3} \end{Bmatrix} = \begin{Bmatrix} 2-(-1)^4 \\ 4 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 4 \end{Bmatrix} \tag{2-15}$$

Step 4

$$\begin{Bmatrix} P_{33T}^{0,4} \\ P_{33C}^{0,4} \end{Bmatrix} = \begin{Bmatrix} 1 \\ 4 \end{Bmatrix} \tag{2-16}$$

second cycle

Step 1

$$\begin{Bmatrix} P_{33T}^{1,1} \\ P_{33C}^{1,1} \end{Bmatrix} = \begin{Bmatrix} 1+(-1)^4 \\ 4 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4 \end{Bmatrix} \tag{2-17}$$

Step 2

$$\begin{Bmatrix} P_{33T}^{1,2} \\ P_{33C}^{1,2} \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4+(-1)^2 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 5 \end{Bmatrix} \tag{2-18}$$

Step 3

$$\begin{Bmatrix} P_{33T}^{1,3} \\ P_{33C}^{1,3} \end{Bmatrix} = \begin{Bmatrix} 2 \\ 5 \end{Bmatrix} \tag{2-19}$$

Step 4

$$\begin{Bmatrix} P_{33T}^{1,4} \\ P_{33C}^{1,4} \end{Bmatrix} = \begin{Bmatrix} 2 \\ 5-(-1)^2 \end{Bmatrix} = \begin{Bmatrix} 2 \\ 4 \end{Bmatrix} \tag{2-20}$$

We can carry on the process to produce the following trace for the bobbin. The process can be carried on to show that the orbit of the bobbin initially at (3,3) is given by:

(3,3), (1,4), (2,4), (4,2), (5,2), (7,4), (8,4), (10,2), (11,2), (13,4), (13,5), (12,3), (10,1), (9,3), (7,5), (6,3), (4,1) (the sequence repeats itself)

In each ordered pair the first element is the track position at the end of the cycle and the second element is the column position.

The motion of the bobbin is periodic. While to trace the yarn it is not necessary to determine the period, the information is useful in tracing the bobbin on the loom and to study the braid cross-section. The knowledge also helps in conserving computer memory when the formulation is used to simulate the machine motion on a computer. The period is

measured by comparing the position of the bobbins at the end of each cycle to the initial position. A full period is braided when the following equation becomes true.

$$\forall(i,j,k), P_{ijk}^{p,4} = P_{ijk}^{0,0} \tag{2-21}$$

Equation (2-21) implies that all the bobbins move to their initial positions.

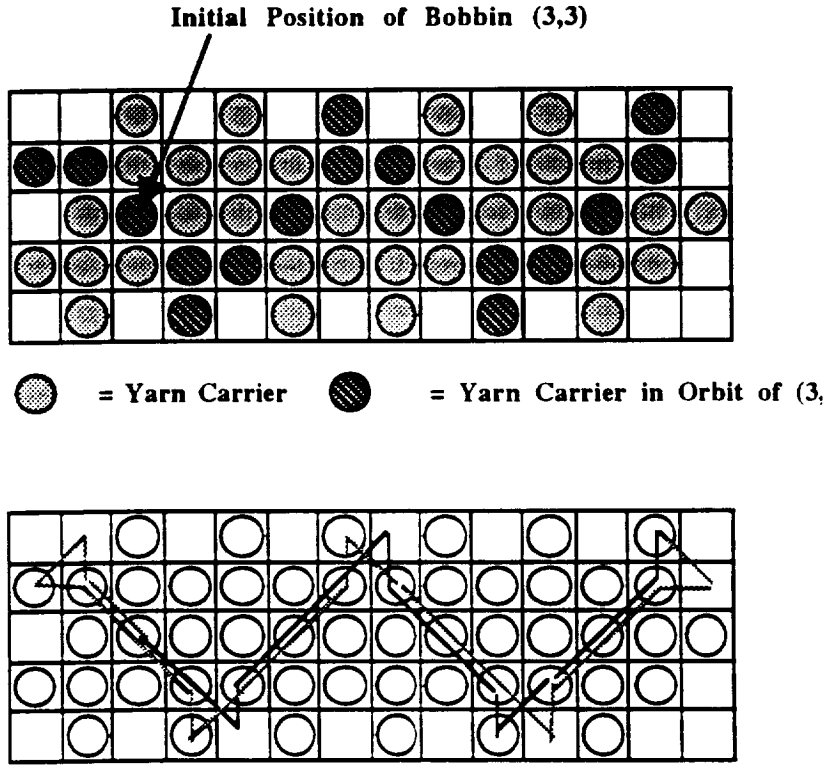


Fig 2.3 Machine Setup of 5×14 loom for 1×1 braiding and the orbit of bobbin initially at (3,3)

We note that the orbit of the element (3,3) do not pass through all possible bobbin positions. Out of 51 possible bobbin positions the orbit passes through 17 of them. If a single yarn passes through all possible bobbin positions, the braid pattern is called fully integrated. In a fully integrated braid, the orbit of a single yarn determines the entire pattern. The braid shown in Figure 2.3 is not fully integrated. To obtain the complete braiding pattern we need to examine the orbit of other elements. Let us examine the orbit of outer elements (3,2) and (3,4). The orbit of (3,2) is given by:

(3,2), (5,4), (6,4), (8,2), (9,2), (11,4), (12,4), (14,3), (12,1), , (11,3), (9,5), (8,3), (6,1), (5,3), (3,5), (2,3), (2,2), (3,2)

The orbit of (3,4) is given by:

(3,4), (4,4), (6,2), (7,2), (9,4), (10,4), (12,2), (13,2), (13,3), (11,5), (10,3), (8,1), (7,3), (5,5), (4,3), (2,1), (1,2), (3,4)

We see that the three orbits are mutually exclusive and they together pass through all 51 possible bobbin position. Therefore for our example the orbit of these three elements together define the complete brading pattern.

As an example of an integrated braid we examine a 5×13 braid. Here m=5, n=13. Again using equations (2-2) to (2-11), we determine the orbit of element (3,3) as follows:

(3,3), (1,4), (2,4), (4,2), (5,2), (7,4), (8,4), (10,2), (11,2), (13,3), (11,5), (10,3), (8,1), (7,3), (5,5), (4,3), (2,1), (1,2), (3,4), (4,4), (6,2), (7,2), (9,4), (10,4), (12,2), (13,2), (11,3), (9,5), (8,3), (6,1), (5,3), (3,5), (2,3), (1,2), (3,2), (5,4), (6,4), (8,2), (9,2), (11,4), (12,4), (12,3), (10,1), (9,3), (7,5), (6,3), (4,1)

As it can be seen from Figure 2.4, the orbit passes through all possible 47 bobbin positions. This also shows that slight change in processing parameters may cause a drastic change in braid pattern.

2.2.5 The Yarn Trace

Once the elements of the position tensor **P** is known, the yarn trace can be calculated. In calculating the yarn trace, it is assumed that a motion L of the loom will cause the yarn at the braiding plane to move a distance Lr. r is defined as the braiding ratio. The takeup is assumed to take place at every second step.

With the above assumptions for yarn starting from bobbin ij, we have:

$$X_{ij}^{c,s} = P_{ijT}^{c,s} . r \qquad (2\text{-}22)$$

Here, $X_{ij}^{c,s}$ is the x-coordinate of the yarn. The origin of the coordinate system is at the braiding plane corresponding to yarn 1,1.

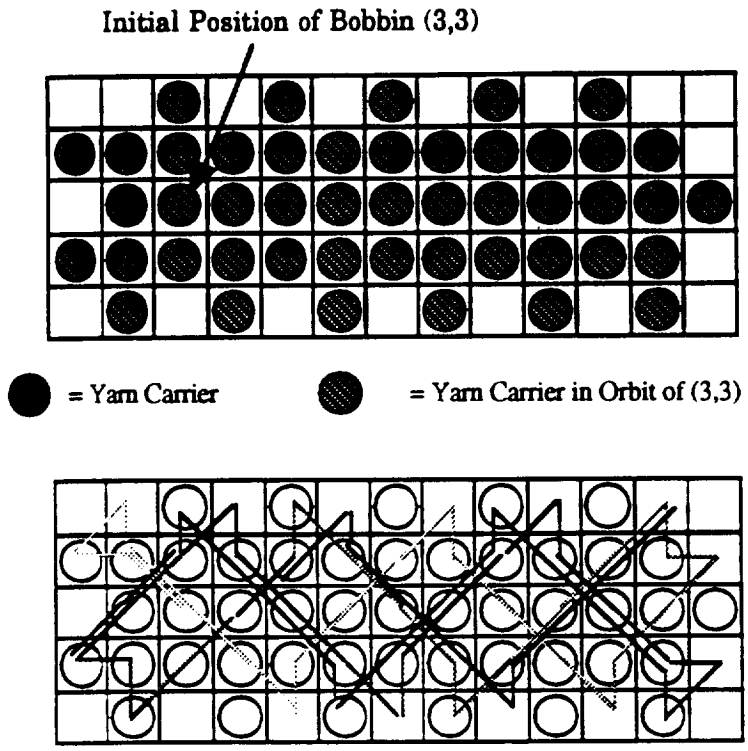


Fig 2.4 Machine Setup of 5×13 loom for 1×1 braiding and the orbit of bobbin initially at (3,3)

Similary,

$$Y_{ij}^{c,s} = P_{ijC}^{c,s}.r \tag{2-23}$$

and,

$$Z_{ij}^{c,s} = \left[2p + \mathrm{Int}\left\{\frac{s}{2}\right\}\right] * \mathrm{pick} \tag{2-24}$$

2.3 Multiaxial Warp Knit.

Knitted fabrics are interlooped structures wherein the kintting loops are preodued by the introduction of the knitting yarn either in the cross machine direction (weft knit) or along the machine direction (warp knit). The unique feature of the weft knit structures is their conformability. But, the most undesirable feature, from the structural reinforcement point of view, of the weft knit structure is their bulkiness which leads to the lowest packing

density, or lowest level of maximum fiber volume fraction compared to the other fabric preforms. Thus, the multiaxial warp knit (MWK) 3-D structures are more promising and they have undergone a great deal more development in recent years.

The MWK fabric systems consist of warp (0°), weft (90°), and bias (±θ) yarns held together by a chain or tricot stitch through the thickness of the fabric, as illustrated in Figure 2.5. The major distinctions of these fabrics are the linearity of the bias yarns; the number of axes; and the precision of the stitching process. The way of introducing the bias yarns is to lay in a system of linear yarns at an angle. Depending on the number of guidebars available and the yarn insertion mechanism, the warp knit fabric can consist of predominately uniaxial, biaxial, triaxial or quadraxial yarns. The latest development, as shown in Figure 2.6, in the impaled MWK is the LIBA system wherein six layers of linear yarns can be assembled in various stacking sequences and stitched together by knitting needles piercing through the yarn layers.

Theoretically, the MWK can be made to as many layers of multiaxial yarns as needed, but the current commercially available machines only allow four layers (the Mayer system) of 0°, 90°, +θ, and -θ insertion yarns, or six layers (the LIBA system) of 2(90°), 0°, 2(+θ), and -θ insertion yarns to be stitched together. All layers of insertion yarns are placed in perfect order each on top of the other in the knitting process. Each layer shows the uniformity of the uncrimped parallel yarns. To ensure the structural integrity, it is clear that the 0° yarns cannot be placed in either top or bottom layer. The insertion yarns usually possess a much higher linear density than the stitch yarns, and are therefore the major load bearing components of the fabric.

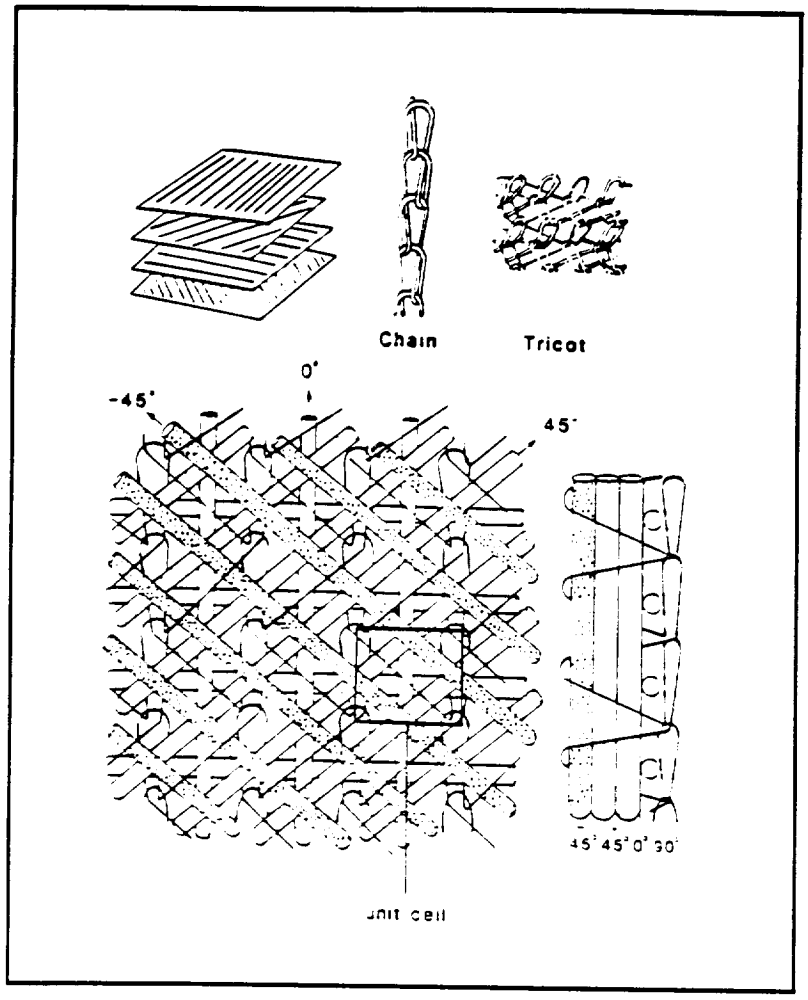


Figure 2.5 MWK fabric systems.

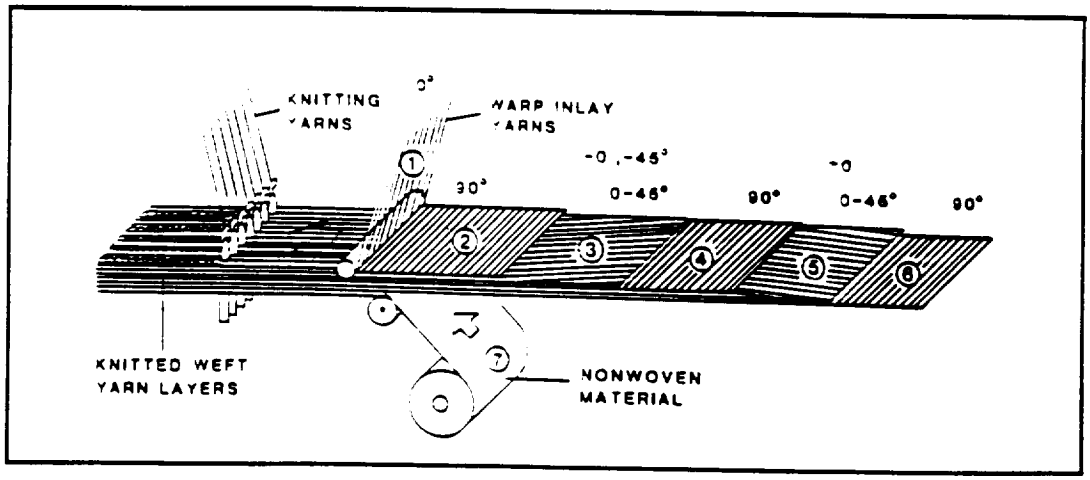


Figure 2.6 MWK LIBA system.

The MWK fabric preforms having four directional reinforcements similar to quasi-isotropic lay-up can be produced in a single step. Besides good handleability and production economics, the MWK fabric preforms also provide the conformability to complex shapes, the flexibility in the principal yarn directions, and the improved through-the-thickness strength. The MWK fabrics have now been used for playing field surfaces, rubber coated fabrics for lifeboats, rescue floats, inflatable sport boats, heavy-duty tarpaulins, geotextiles and filter fabrics, and automotive air bags.

The key geometric parameters of the MWK fabric preforms, which affect the reinforcement capability and the composite processability, include the number of yarn axis, the orientation of bias yarns, total fiber volume fraction, pore size and pore distribution, and percentage of stitch fibers to total fiber volume. The process variables adjustable to control the MWK micro-structure include the type of knit stitch, the ratio of stitch-to-insertion yarn linear density, the orientation angle of bias yarns, and the thread count. The concept of a unit-cell is used to establish the relationship between the geometric parameters and process variables.

2.3.1 Knitting Process

The principal mechanical elements used in knitting are needles, which form the loops to interlace the linear insertion yarns. Therefore, knitting process can be fully understood if the motion of the needles is described. The loops in knitted fabrics are formed essentially on a very similar principle. Following Thomas [2.3], the looping process is demonstrated for a single latch needle by the consecutive steps shown in Figure 2.7. The general knitting action of a latch-needle machine can be found in [2.4].

Consider the needle which has at its stem a loop already fromed during the course of the knitting process, as in Figure2.7(a). A thread is then placed under the hook of the needle. The loop is restrained in its position whereas the needle is allowed to move through it. As the needle moves downward, the existing loop will push the latch and close the hook (Figure 2.7c). When the top of the hook reaches the level of the existing loop (Figure 2.7d), this loop is pulled out of the way by the yarn tension. Then as the needle moves upward again, the thread in the hook opens up the latch, and it becomes the next 'existing' loop. More loops are generated as the process repeats.

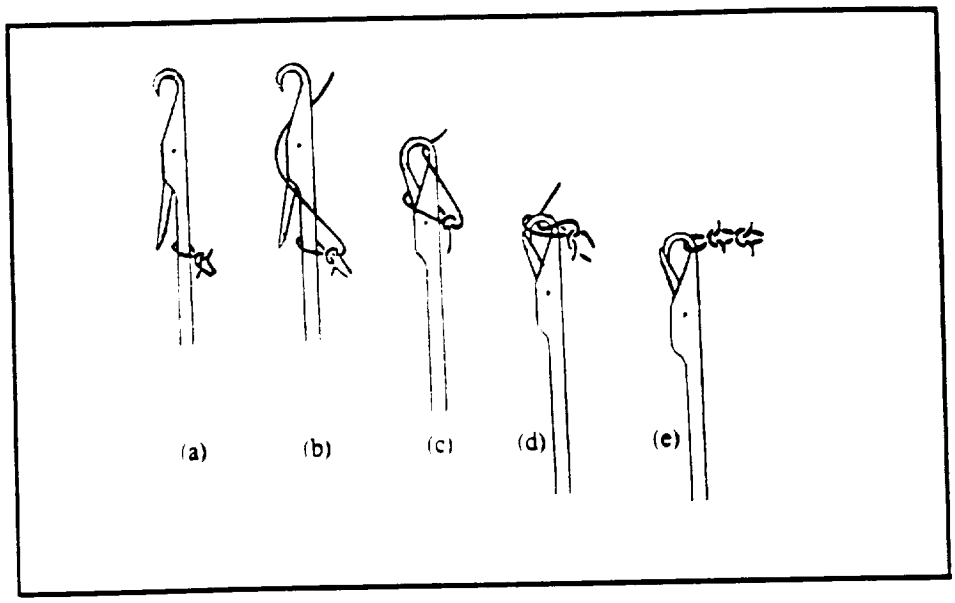


Figure 2.7 The needle cycle in knit fabrics.

Generally, the warp knitted loop structur is made of two parts. The first one is the loop itself, which is fromed by the yarn being wrapped around the needle and drawn through the previous loop, as described above. This part of the structure is called an overlap. The second part is the length of yarn connecting the loops, which is called an underlap. It is formed by the shrogging movements of the ends across the needles. Since the underlap is connected to the root of the loop, it causes, due to warp tension, an inclination to the loop structure.

Two different lap forms are used in warp knitting, depending on the way the yarns are wrapped around the needles to produce an overlap. When the overlap and the next underlap are made in the same direction, an open lap is fromed, shown in Figure 2.8(a). If, however, the overlap and the following underlap are in the opposition to one another, a closed lap is formed, shown in Figure 2.8(b).

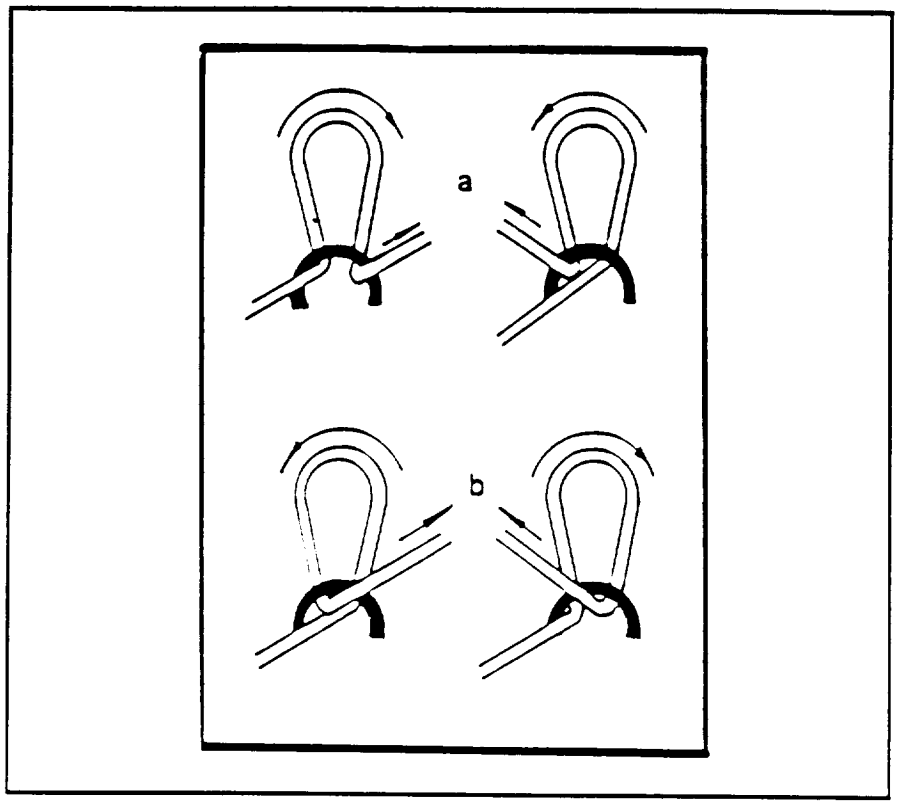


Figure 2.8 Open and closed lap configurations

For a chain (pillar) stitch, it is formed when a needle is being lapped continously by the same guide. Since the guide bar does not lap the adjacent needles, there are no sideways connections. The chain lapping movement can be open (see Figure 2.9), closed (see Figure 2.10) or can be a combination of closed and open laps. The more common open lap chain construction, shown in Figure 2.11, is fromed when the guide laps the needle alternately from the right and the left. The chain notations, as derived from Figure 2.9, are 0-1 for the first course and 1-0 for the next. To produce a closed lap chain, the guide has to lap the needle continously in the same direction and the chain notations are 0-1 for all courses, as illustrated in Figure 2.10.

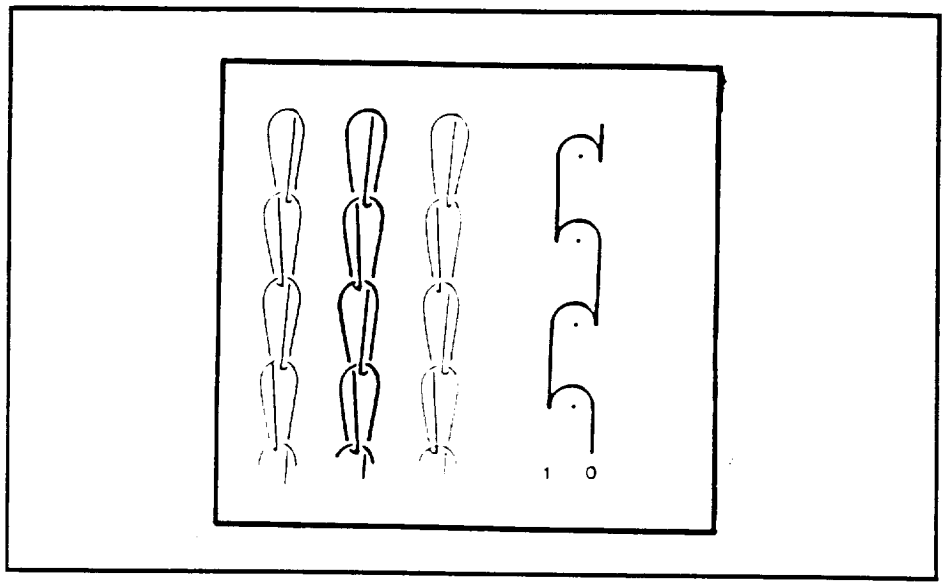


Figure 2.9 An open-lap chain stitch.

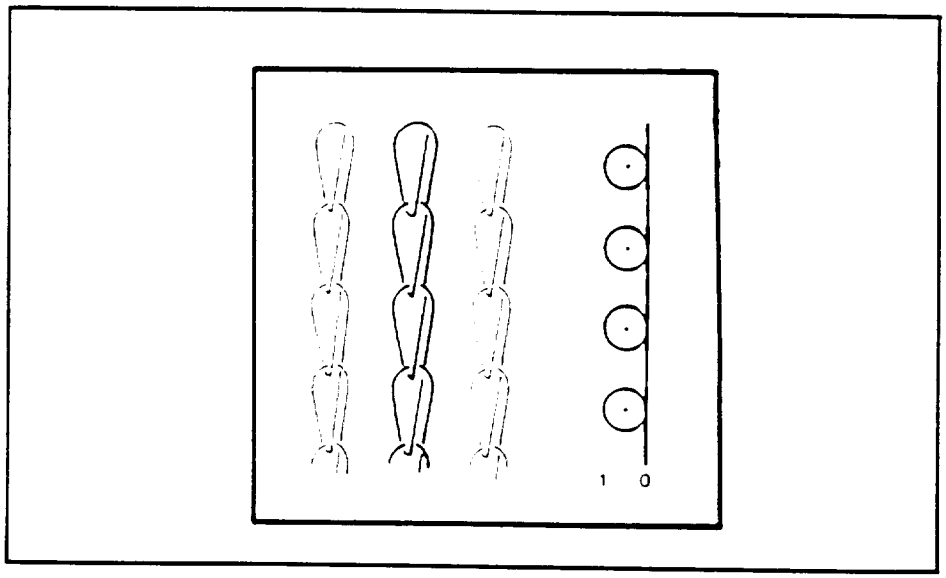


Figure 2.10 A closed-lap chain stitch.

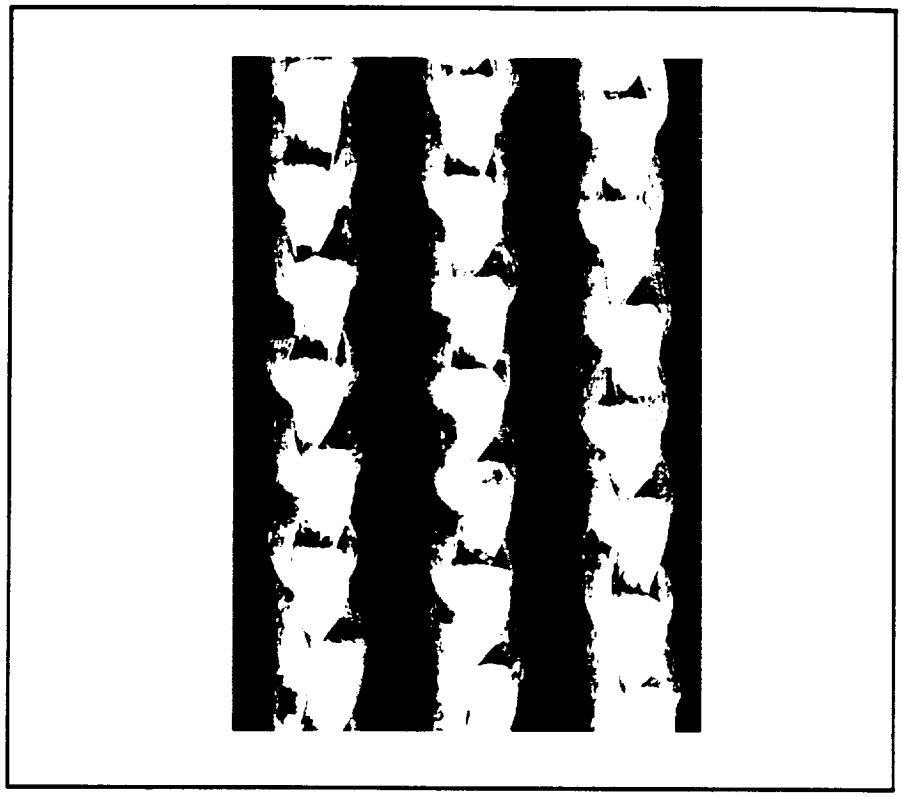


Figure 2.11 An open-lap chain stitch construction

For the multi-axial warp knits(MWK), straight warps, straight filling and off-axis lay-ins are introduced and the stitch yarn is looped over the intersection of filling yarn and off-axis yarns. As shown in Figure 2.5, the unit cell of the MWK includes warp (0°), weft (90°), and bias ($\pm\theta$) yarns held together by a chain or tricot stitch through the thickness of the fabric. The size of the unit cell depends on the orientation of off-axis yarns and the diameter of the insertion yarns. In this report, the chained MWK is mainly focused, and the tricot MWK can be studied in a similar way.

2.3.2 Stitch Loop Model

As mentioned previously in this section, the stitch loop is formed by needle motion. But, it is very difficult to trace the stitch yarn path in the 3-D space when it is guided by the needle. Unlike the machine motion of 3-D braid, the relationship between machine motion and stitch yarn path in MWK is imposibly established. Therefore, assumptions have to be made in order to describe the stitch yarn path. The following points should be considered before a loop model is to be built.

1. The shape of the loop, loop inclination and fabric weight may change slightly without changing the course and wale counts.

2. The configuration of the yarn within the knitted cell is affected by a great number of variables such as yarn properties, warp tensions, ta-up tensions, yarn lubrication , etc.

It is proposed that the loop shape of the fabric in the machine state is more likely to be determined by the physical pull of the take-up mechanism and the accommodation of the root configuration, than by the bending forces. Taking these facts into account, the following loop model for the fabric in the machine state, shown in Figure 2.12, is suggested.

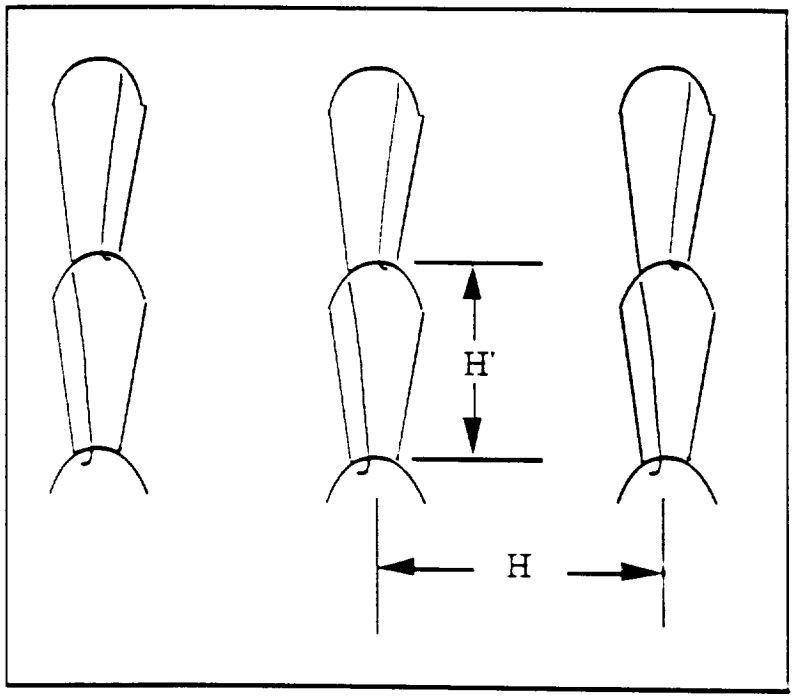


Figure 2.12 The "machine state" loop model

The model can be divided into three parts:

1. The loop shape which is illustrated in Fig.2.13 can be described as the loop's head and the straight arms. The loop head is on

horizontal plane, z = 0, while the loop arm is in a 3-D space.

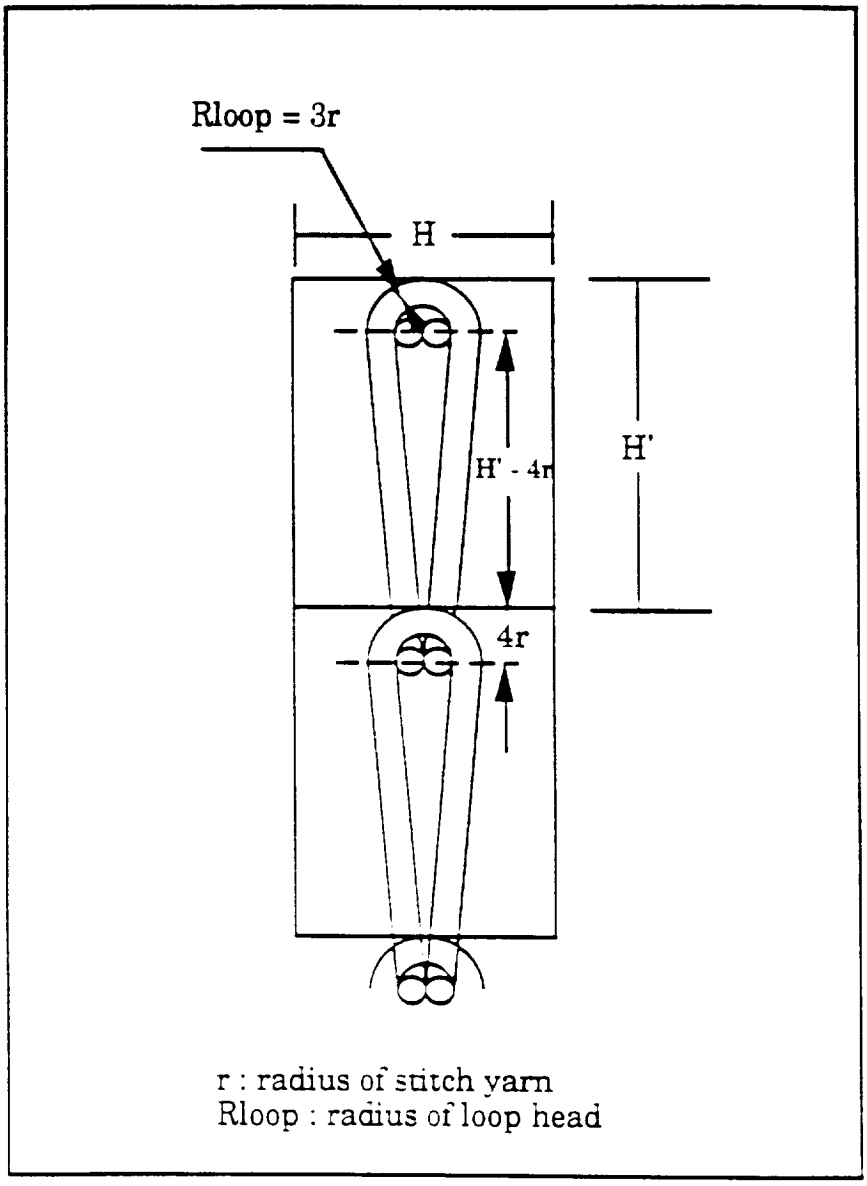
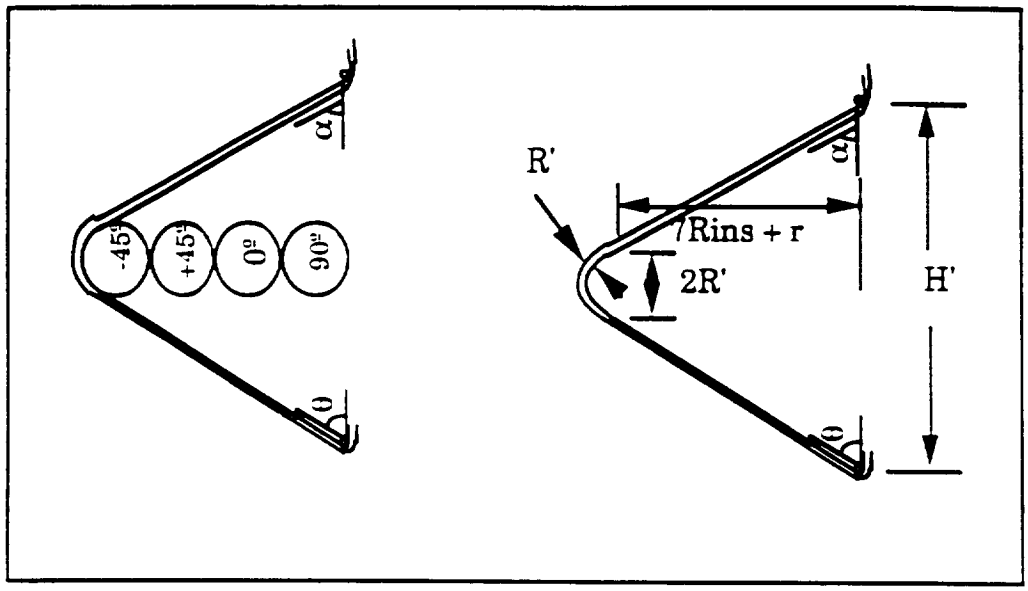


Figure 2.13 The loop part of the 'machine state' loop model.

loop's head = $2\pi Rloop = 6\pi r$

$$\text{loop's arm} = 4\sqrt{(H' - r)^2 - 4r^2} \qquad (2\text{-}25)$$

2. The loop-over, which is the loop over insertion yarns, can be described in the following figure 2.14:



Rins : radius of the insertion yarn

r : radius of the stitch yarn

R' : radius of the circular arc, which partially wrap around the off-axis insertion yarn

R' = Rins/cos(45°) = $\sqrt{2}$ Rins

Figure 2.14 The loop over the insertion yarns from the loop model

loop-over's head $= 2\pi R' = 2\sqrt{2}\pi Rins$

$$\text{loop-over's arm(entering)} = 2\sqrt{(\sqrt{2}Rins + r)^2 + (\frac{H'}{2} - 4r - R')^2} \quad (2\text{-}26)$$

$$\text{loop-over's arm(leaving)} = 2\sqrt{(\sqrt{2}Rins + r)^2 + (\frac{H'}{2} - 4r - R')^2}$$

3. The details of the root of the loop are illustrated in Figure 2.15. The first kind of root for the yarn entering the loop-over can be calculated as:

$$2\pi r\left(\frac{\pi - \theta}{\pi}\right) = 2\pi r\left(1 - \frac{\theta}{\pi}\right) \tag{2-27}$$

where $\theta = \tan^{-1}\left(\frac{7Rins}{\frac{H'}{2} + 4r - \sqrt{2}\,Rins}\right)$

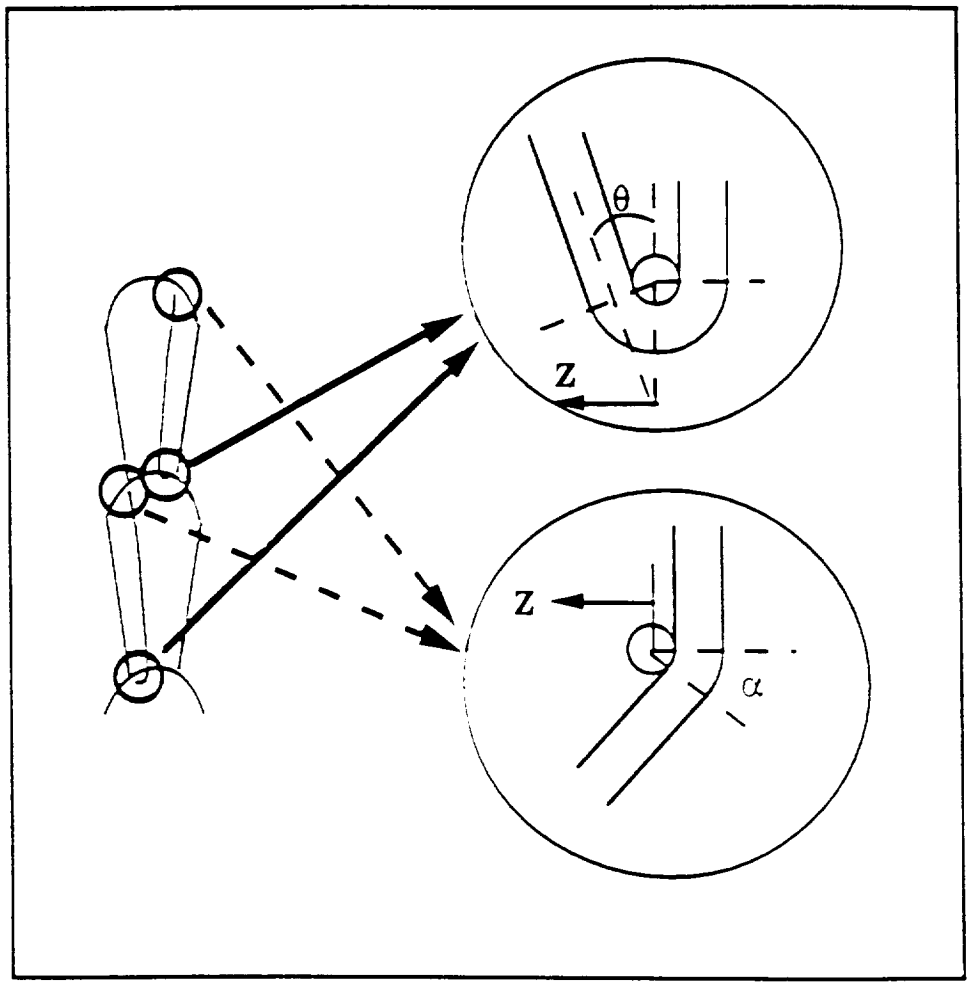


Figure 2.15 The yarn root configuration

The second kind of root for the yarn leaving the loop-over can be calculated as:

$$2\pi r\left(\frac{\alpha}{\pi}\right)$$

where $\alpha = \tan^{-1}\left(\frac{7Rins}{\frac{H'}{2} - 4r - \sqrt{2}\,Rins}\right)$

The amount of yarn in the root of the loop is therefore:

$$\text{loop's root} = 2\pi r\left(\frac{\alpha}{\pi}\right) + 2\pi r\left(1 - \frac{\theta}{\pi}\right)$$

Hence, the complete model can be described as:

$$L = 6\pi r + 4\sqrt{(H' - r)^2 - 4r^2} - 2\sqrt{2}\,\pi Rins + 2\pi r\left(\frac{\alpha}{\pi}\right) + 2\pi r\left(1 - \frac{\theta}{\pi}\right)$$

$$+ 2\sqrt{(7Rins + r)^2 - \left(\frac{H'}{2} + 4r - R'\right)^2} \qquad (2\text{-}28)$$

$$+ 2\sqrt{(7Rins + r)^2 - \left(\frac{H'}{2} - 4r - R'\right)^2}$$

By applying the loop model stated above, the entire shape of the stitch yarn can be described provided that processing parameters are given. Thus, the key points of the unit cell of a MWK preform, including stitch yarns and insertion yarns, can be obtained through a geometric analysis. The details of how to acquire the key points mathematically will be presented in Chapter three.

2.4 References

2.1. Ko, F. K., 1989, "Three-Dimensional Fabrics for Composites", *Textile Structural Composites*, Vol.3, Elsevier, Amsterdam, the Netherlands, pp. 129-171.

2.2. Pastore, C. M. and Ko, F. K., 1990, "Modeling of Textile Structural Composites, Part I: Processing-science Model or Three-dimensional Braiding", *J. Text. Inst.*, Vol. 81, pp. 480-490.

2.3 Thomas, D. G. B.(1971) An Introduction to Warp Knitting, Merrow, Watford, UK

2.4 Raz, S. (1987) Warp Knitting Production, Melliand Textilberichte GmbH, Germany.

Chapter - 3 Geometric Modeling

In this chapter, the theoretical framework of computer aided geometric modeling for the structural analysis of textile reinforced composite materials is presented. Based on the processing model developed in Chapter 2, the illustration of generating key points for each yarn path is presented for 3-D braid and Multiaxial Warp Knit composites, respectively.

3.1 Theoretical Background

The first step in the construction of a geometric model of a textile preform is to model the preform processing technique. The machinery which forms the textile preform dictates the geometry of the yarn path. By proper modeling of the process, it is possible to generate the basic information associated with the yarn path. The modeling of the process has been discussed in detail in Chapter 2.

The algorithm is developed such that the designer can enter concise information about the machine parameters and have the algorithm output a set of "knots" which describe the yarns and yarn paths within the fabric. The "knots" are a sequence of points defining the center of the yarn in such a way that the entire yarn can be regenerated from this finite set.

3.1.1 Yarn Path Modeling

Given a set of knots from the modeling of the textile structure, it is now necessary to predict the geometric behavior of the yarn between these points. The yarn path will be dictated by the material properties, the cross-sectional geometry of the yarn, and the motion of the yarn relative to the other yarns.

The material properties of the yarn are modeled in terms of the minimization of strain energy on the yarn element. Strain energy is minimized through the incorporation of a B-spline function to generate the path of the yarn. The basic assumption of the B-spline is to minimize the term [3.1]

$$\int_0^L r''ds \qquad (3\text{-}1)$$

where, L= total length of the yarn

$$r'' = \frac{d^2 r}{ds^2}$$

r = g(s) = spline function of the yarn path

and, s = path length.

since r'' is closely related to κ (the curvature), the minimization of this term minimizes curvature, and hence the bending strain (ε = E κ) of the yarn. Additionally, the use of the Bspline minimizes the twist in the yarn [3.1] In this work a constant cross-section has been employed, which twists according to the path of the yarn as defined by the B-spline.

Given a set of points describing the center of a yarn i, $\left\{ P_j^i \right\}_{j=1}^{n}$, it is necessary to construct additional control points to form the B-spline. These control points are formed in such a way as to provide differential continuity for the yarn path. To construct the curve around point $\mathbf{P}_j^i$, four control points are needed, as illustrated in Figure 3.1:

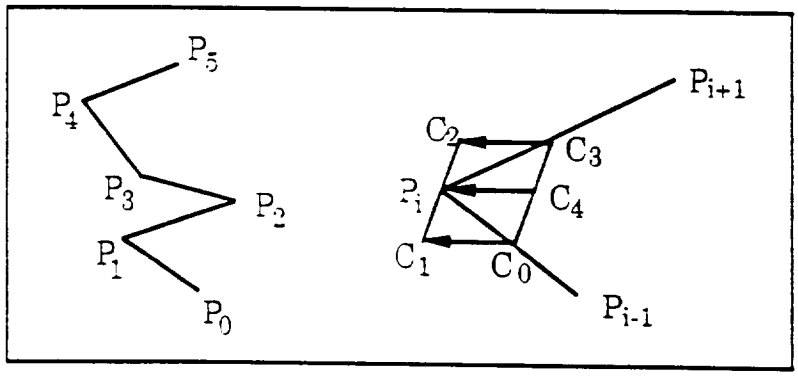


Figure 3.1. Construction of Control Points for B-Spline Generation .

To assist in the construction of the points, a fifth point is introduced, which serves only to make the calculations more readable. These points can be constructed in groups. The first two control points are the mid-points of the vectors connecting the point to its predecessor and successor:

$$\mathbf{C}_0^i = \frac{P_i + P_{i-1}}{2} \qquad (3\text{-}2)$$

$$C_3^i = \frac{P_{i+1} + P_{i-1}}{2} \tag{3-3}$$

The mid-point of the vector connecting these two points is not a control point, but is useful for the construction, and is defined as:

$$C_4^i = \frac{C_3^i + C_0^i}{2} \tag{3-4}$$

The remaining two control points are formed so as to be parallel to the vector formed between C_4^i, and p_i. They can be given as:

$$C_1^i = C_0^i + P_i - C_4^i \tag{3-5}$$

$$C_2^i = C_3^i + P_i - C_4^i \tag{3-6}$$

This construction guarantees that the point P_i is collinear with C_1^i and C_2^i, and in fact is the midpoint of the corresponding vector. The value of this definition is that it provides interpolation of the point P_i in the construction of the B-spline [3.2].

Having constructed the control points, it is now possible to generate the path of the center of the yarn using the B-spline. The path of the yarn is given as:

$$r(t) = \frac{\sum_{i=0}^{n} B_{i,k}(t) w_i C_j^i}{\sum_{i=0}^{n} B_{i,k}(t) w_i} \tag{3-7}$$

where $r(t)$ = yarn spatial path as a function of arc length

w_i = weighting function

and $B_{i,k}(t)$ = B-spline of order k

The B-spline is defined as:

if,

$$t_j \le t < t_{j+1}$$

$$B_{j,k}(t) = 1$$

else,

$$B_{j,k}(t) = \frac{t - t_j}{t_{j+k} - t_j} B_{j,k-1}(t) + \frac{t_{j+k+1} - t}{t_{j+k+1} - t_{j+1}} B_{j+1,k-1}(t) \qquad (3\text{-}8)$$

Since the B-spline is constructed using four control points, in order to achieve interpolation of the knots, the order of the spline is 3 (k=3).

The B-spline, r(t) now describes the path of the center point of the yarn as it moves from point to point within the preform. With the construction of an analytical function describing the path, it is now possible to identify certain critical parameters associated with the mechanical response of the yarn. The twisting angle on the yarn is given by

$$\theta(s) = \cos^{-1}(v_0 \bullet v_1) \qquad (3\text{-}9)$$

where $v_0 = (P_{i+1} - P_i) \otimes (P_i - P_{i-1})$

$v_1 = (P_{i+2} - P_{i+1}) \otimes (P_{i+1} - P_i)$

as illustrated in Figure 3.2.

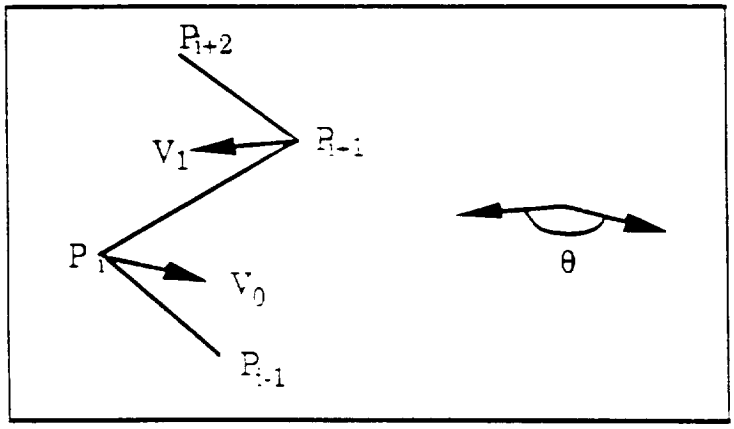


Figure 3.2 Calculation of Twist Angle in Yarn Path.

The twist on the yarn between two points can be given as:

$$T = \frac{\int_0^L \theta(s)\, ds}{L} \qquad (3\text{-}10)$$

where, T = twist per unit length, or torsion.

Similarly, the bending on the yarn path can be given by

$$\cos(\phi) = (P_{i+1} - P_i) \bullet (P_i - P_{i-1}) \tag{3-11}$$

where ϕ = bending angle, as shown in Figure 3.3.

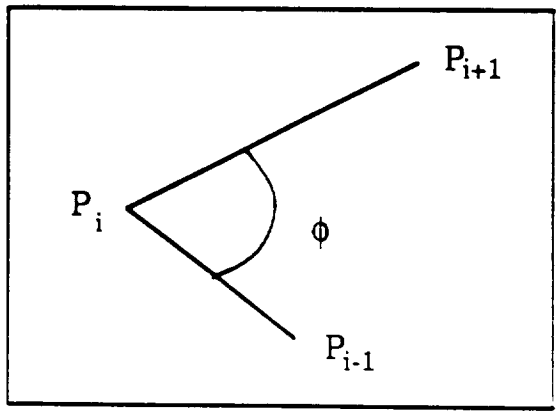


Figure 3.3. Calculation of Bend Angle in Yarn Path.

With these relationships, the basic geometric properties of the yarn center-line are known, and it is possible to construct the three dimensional form of the structure.

3.1.2 Solid Modeling of the Yarn

The solid modeling of the yarn is achieved by sweeping a yarn cross-section along the center-line of the yarn as determined using the algorithm described above. Since the typical yarn does not have a circular cross-section, it is necessary to identify the orientation of the cross-section during the sweeping process. An elliptical cross-section has been chosen to represent a typical yarn, and has been shown to be a reasonable representation.

The ellipse is actually an n-gon resembling an ellipse to an arbitrary degree. The points on the cross section, e_j, are calculated as

$$e_j = \begin{pmatrix} a\cos\frac{j\pi}{n} \\ b\sin\frac{j\pi}{n} \end{pmatrix} \tag{3-12}$$

In this way, a surface representation of the physical yarn can be constructed. The orientation of the ellipse is normal to the yarn path, and the major axis of the ellipse is twisted in accordance with the yarn path twist. An initial orientation of the ellipse is

defined for the yarn at z=0. As twist is identified in the yarn path, the major axis is rotated at the same amount.

The yarn now consists of a surface representation. This surface can be rendered in color or shading for informational display. Any of the suitable shading techniques work well with this structure since it now consists of connected polygons. It is only necessary to identify the viewpoint, the light source, and the reflectivity of the surface. The result is a color shaded rendering of the individual yarns within the fabric.

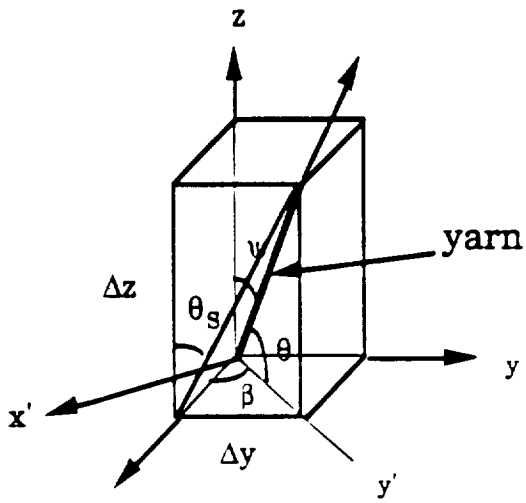
3.2 Application to the Braiding Model

The processing model developed in Chapter 2 can be used to develop the individual motions of the tracks and columns in the braiding machine. Based on the position tensor operations, the positions of a bobbin can be identified for each cycle as the braiding process progresses. The positions stand for the projection into the x-y plane of the path of one yarn in a typical braiding machine.

The $yarn_0(i,j)$ stands for the initial position of the yarn at the i^{th} track and j^{th} column in the loom. The initial position of the yarn(i,j) is $(X_0(i,j), Y_0(i,j), Z_0(i,j))$, where $Z_0(i,j) = 0$. From Equations (2-3) to (2-11), the position of the yarn(i,j) is defined by $X_k(i,j)$ and $Y_k(i,j)$ after k cycles of track/column motions. Thus the sequence of points $\{(X_0(i,j),Y_0(i,j), Z_0(i,j)),(X_1(i,j),Y_1(i,j), Z_1(i,j)), \ldots ,(X_k(i,j), Y_k(i,j), Z_k(i,j))\}$, where $Z_k(i,j) = k \cdot \Delta z$, describes the path of the yarn(i,j). Δz is the distance of each pick, which is determined by the braiding angle and the ratio of track/column movement. The sequence, $\{P_k = (X_k,Y_k,Z_k)\}$, represents the spatial position of key points in a yarn path, and these points can be used as the knots for the three dimensional braid.

These key points are the basis of generating the yarn path using the B-splines as discussed earlier. The geometric model is complete when we have all the key point information on the yarns in the braid. The B-spline algorithm is incorporated in a computer code which is developed using C language and SunView graphics on a SUN 3/160. The modeling program and its user interfaces is described in Chapter 6.

The shape of the yarn cross-section in a 3-D braid can be determined based on the geometric parameters, processing variables and following assumptions: (1). circular yarns with radius, r; (2). no interactions between yarns. Figure 3.4 shows a single yarn in space with surface angle θ_s after 1/1 track/column motion.



θ_S= surface braiding angle

ψ = braiding angle

Figure 3.4 The orientation of a yarn in space.

Δy is the distance between two adjacent key points in the y-direction and its value is equal to the braiding ratio, r, as mentioned in Chapter 2. The distance of each pick, Δz, is calculated by:

$$\Delta z = \Delta y \cdot \cot(\theta_S) = r \cdot \cot(\theta_S) = r \cdot \tan(\psi) \cdot \cos(\beta)$$

the angle θ can be calculated by the following relation:

$$\theta = \tan^{-1}\left[\frac{\cos(\beta)}{\tan(\theta_S)}\right]$$

The inclined angle ψ, or, braiding angle can expressed as:

$$\psi = \frac{\pi}{2} - \theta = \frac{\pi}{2} - \tan^{-1}\left[\frac{\cos(\beta)}{\tan(\theta_S)}\right]$$

For circular yarns, the yarn cross-section becomes elliptical on a cut plane parallel to x-y plane as illustrated in Figure 3.5:

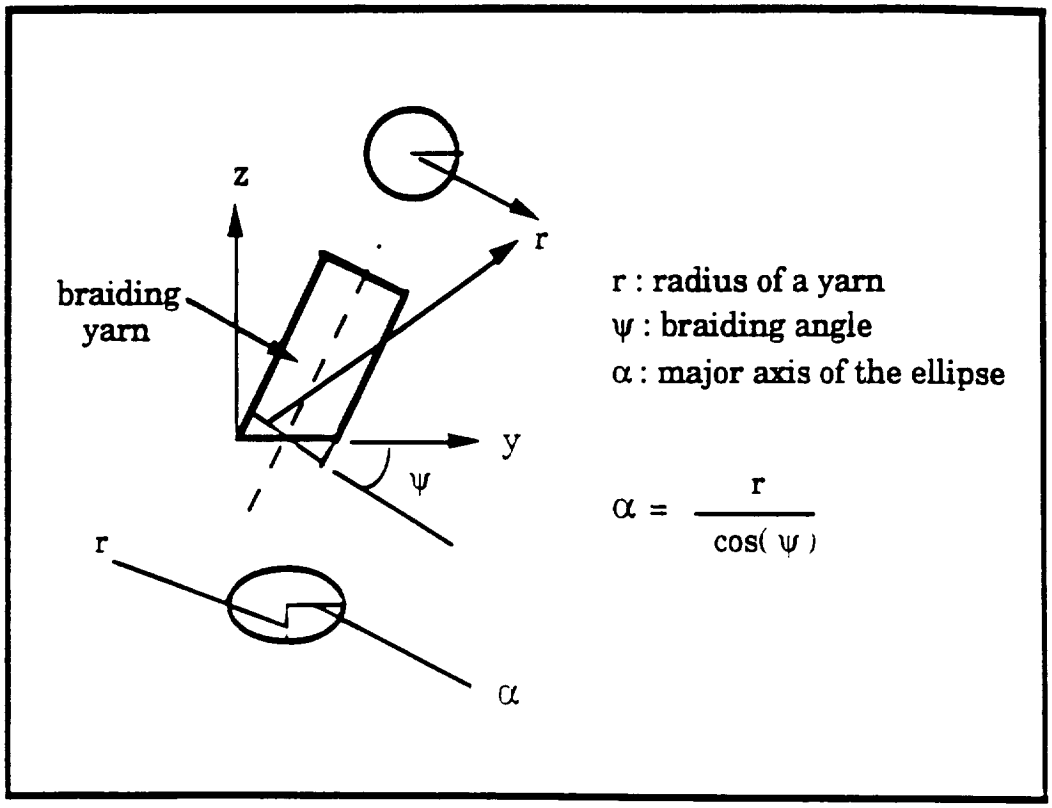


Figure 3.5 Cross-section of a yarn on different planes.

3.3 Application to Multiaxial Warp Knit (MWK)

Theoretically, a MWK can be made to as many layers of multiaxial yarns as needed, but the current commercially available machines only allow four to six layers of 0°, 90°, +θ, and -θ insertion yarns to be stitched together. Based on the processing model presented in Section 2.3, the MWK structure consists of four basic components: warp(0°) yarns, weft (90°) yarns, bias (±θ) yarns, and stitch yarns through the thickness of the fabric. The dimensions of a unit cell of MWK composites can be expressed as H x 2H' x Hz, where H is the spacing between two o° insertion yarns, H' is equal to H(cotθ), and Hz is the thickness of the unit cell. The geometric relationships of the unit cell is illustrated in Figure 3.6.

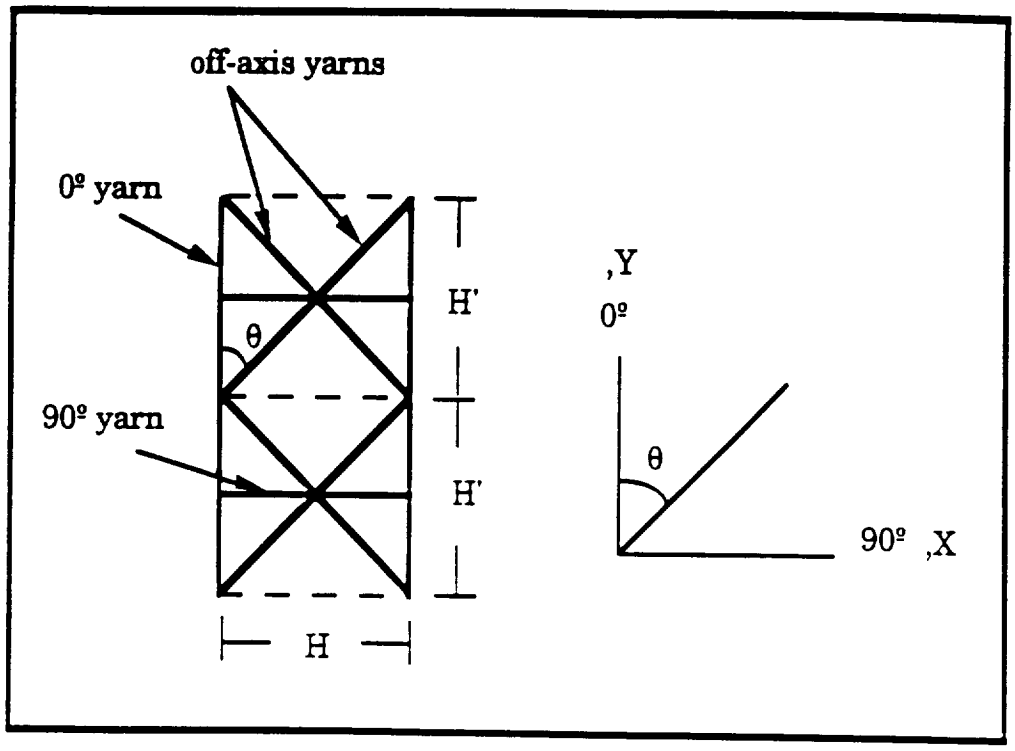


Figure 3.6(a) The planar dimension of a unit cell of MWK composites

As can be seen in Figure 3.6(a), the 0° insertion yarns are H apart, while 90° insertion yarns are H' apart. The off-axis insertion yarns travel through the diagonal orientation of the H x H' rectangle in the through-thickness projection. As in the thickness direction shown in Figure 3.6(b), the z-coordinate of the center point of each insertion yarn are listed in Table 3-1. Therefore, the key points for representing insertion yarns can be determined.

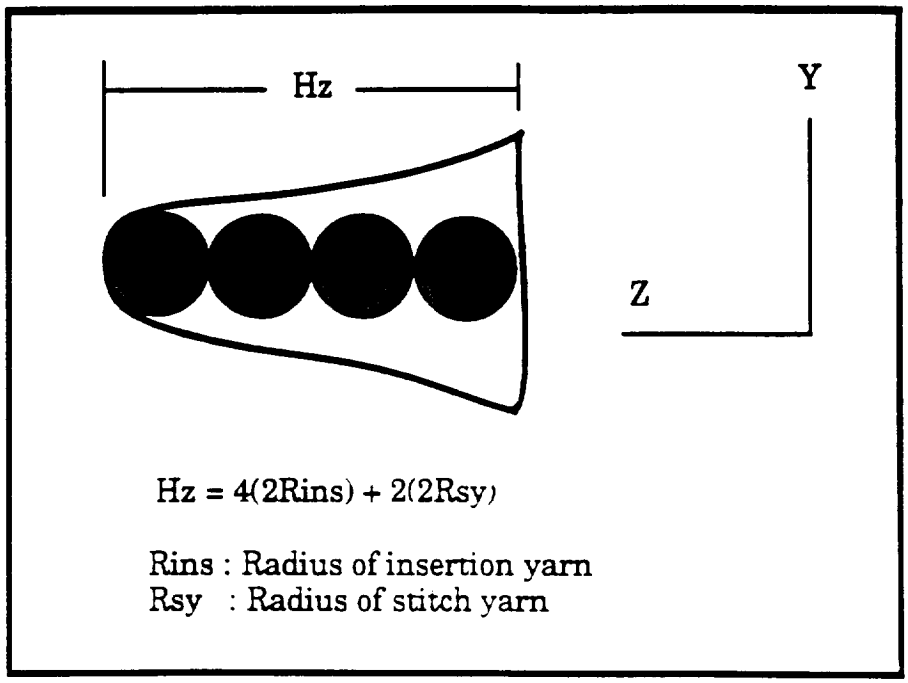


Figure 3.6(b) The through-thickness dimension of a unit cell of MWK composites

Table 3-1 Z-coordinate of the center point of each insertion yarn

insertion yarn	z-coordinate
90°	Rins + Rsy
0°	Rins + 3Rsy
+θ	Rins + 5Rsy
-θ	Rins + 7Rsy

For the stitch yarn, based on the processing model given in Section 2.3, twenty-seven key points are identified in order to represent the stitch yarn within a unit cell. The location of each key point is marked and shown in Figure 3.7.

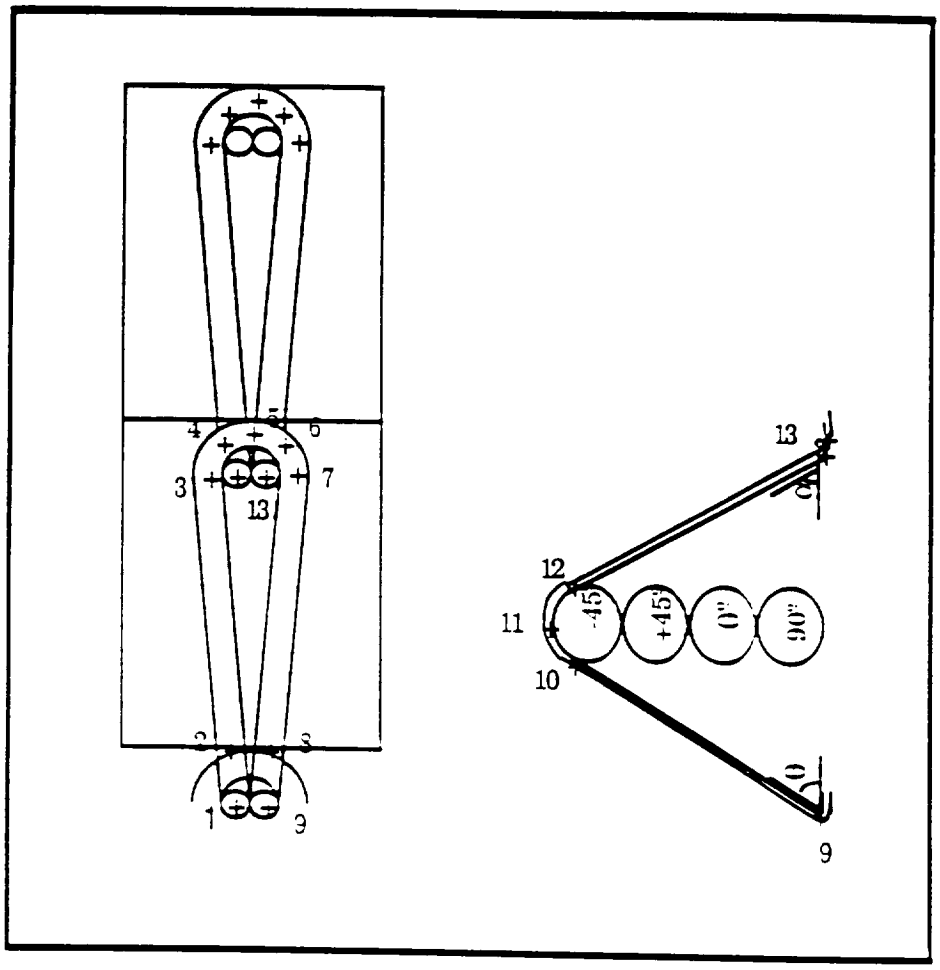


Figure 3.7 Location of each key point in a unit cell.

These key points are usually located at the intersections of two kinds of curves, for instance, the intersection of straight line and arc (loop). The coordinates of each key point can be expressed in terms of the processing parameters, such as H, H', Rinsert, Rloop, and Rsy. The origin of the coordinate system is located at the lower-left corner of the unit cell. For example, at point #1, it can be readily calculated that

$$\text{x-coordinate} : \frac{H}{2} - Rsy$$

y-coordinate : - Rsy - Rloop

For another example, at point #2, the geometric relationship is shown in Figure 3.8.

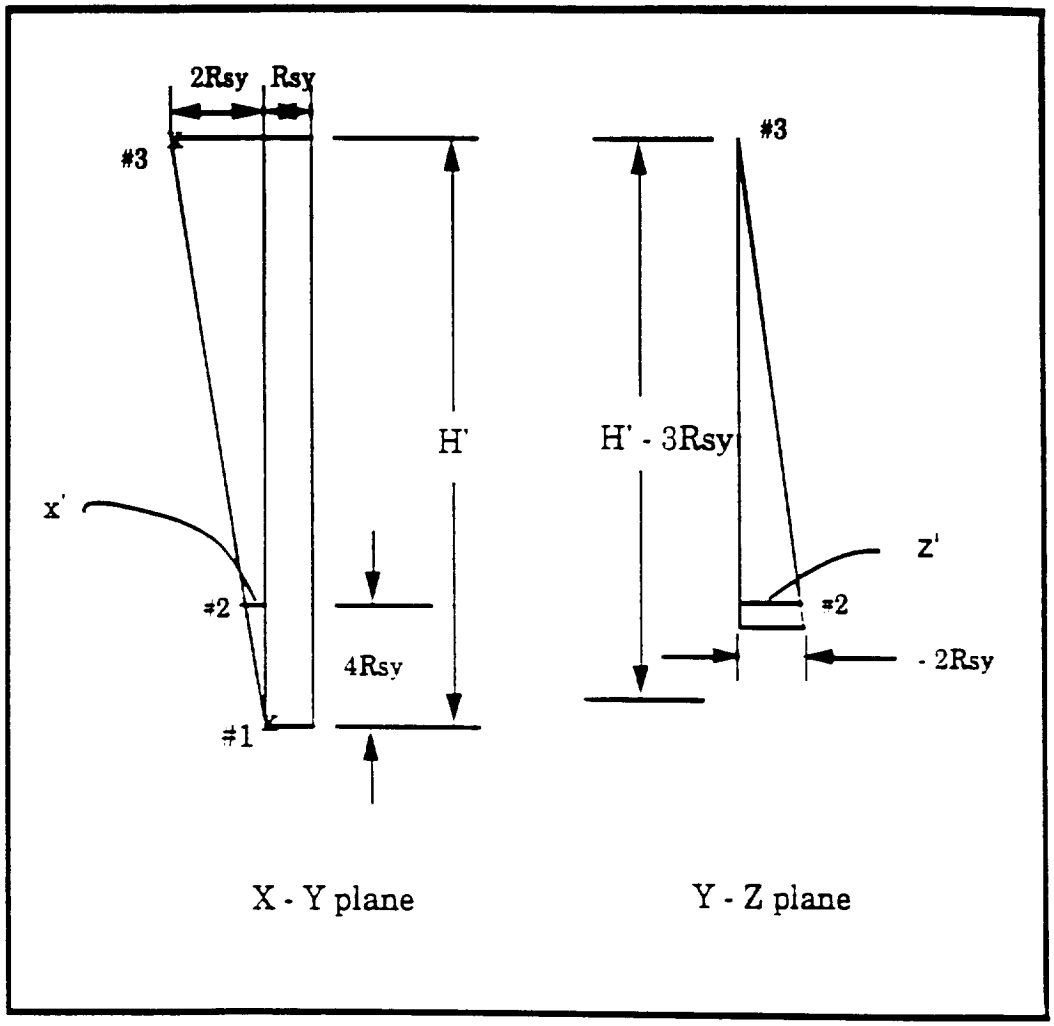


Figure 3.8 Geometric relationship for key point #1, #2 and #3.

$$x' = 2Rsy\ (\frac{4Rsy}{H'})$$

Thus, the x-coordinate is equal to $\frac{H}{2} - x' - Rsy\ [= \frac{H}{2} - Rsy - 4Rsy(\frac{2Rsy}{H'})\ ]$

The distance z' can be calculated in the following trigonometric operation:

$$z' = (-2\ Rsy)(\frac{H' - 4Rsy}{H' - 3Rsy}) = -2Rsy + \frac{Rsy}{H' - 3Rsy}$$

Thus, the coordinates of the point #2

$$\text{x - coordinate :} \quad \frac{H}{2} - Rsy - 4Rsy\left(\frac{2Rsy}{H'}\right)$$

$$\text{y - coordinate :} \quad 0$$

$$\text{z - coordinate :} \quad -2Rsy + \frac{Rsy}{H' - 3Rsy}$$

For another example, Figure 3.9 shows the geometric relationship between the key point #8 and point #9.

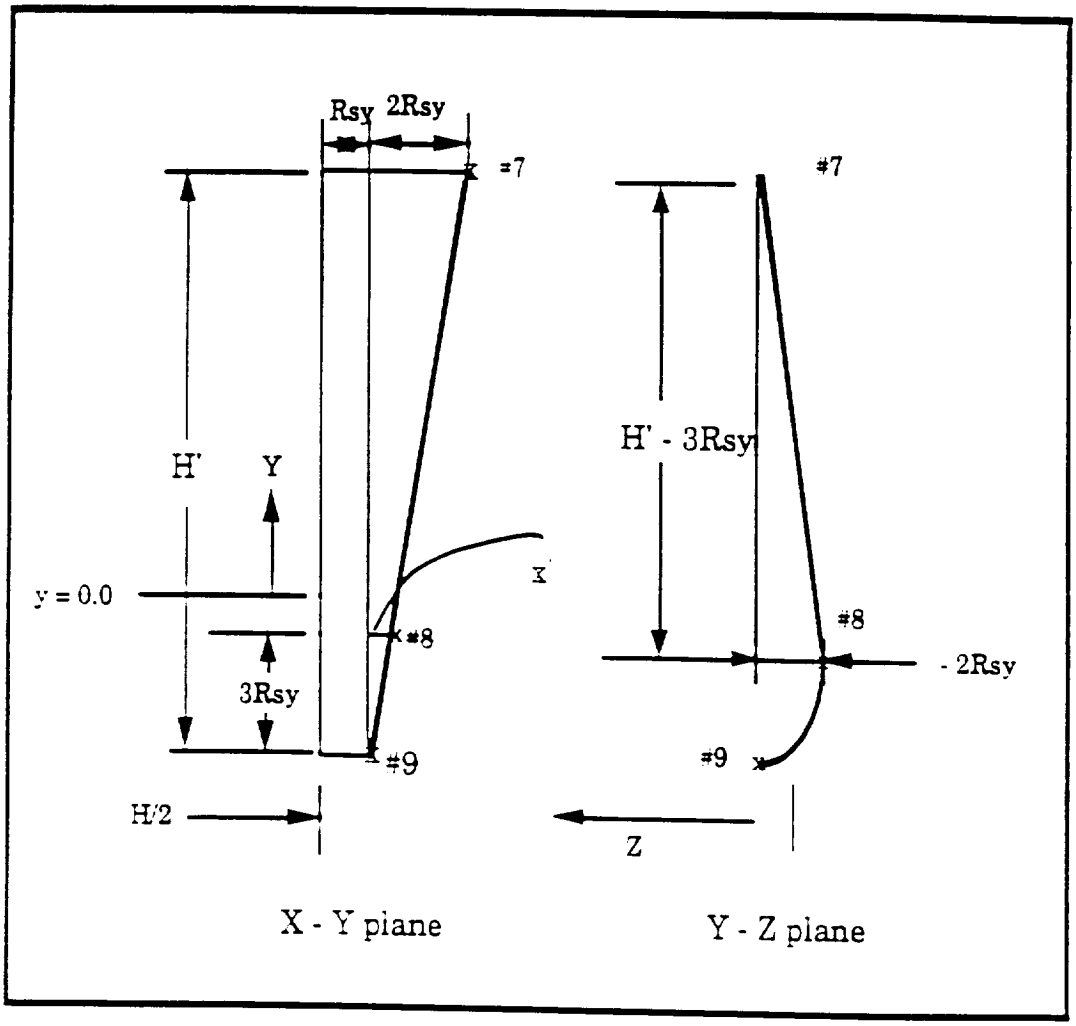


Figure 3.9 Geometric relationship for point #8 and #9.

The distance x' can be determined by triangular relationship:

$$x' = 3Rsy\left(\frac{2Rsy}{H'}\right)$$

Thus, the x-coordinate of key point #8 is

$$x = \frac{h}{2} + x' = \frac{h}{2} + 3Rsy\left(\frac{2Rsy}{H'}\right)$$

For the y-coordinate, the enlarged geometric relationship is shown in Figure 3.10. The horizontal distance, d, can be determined from triangular relationship:

$$d = 3Rsy \tan(\theta)$$

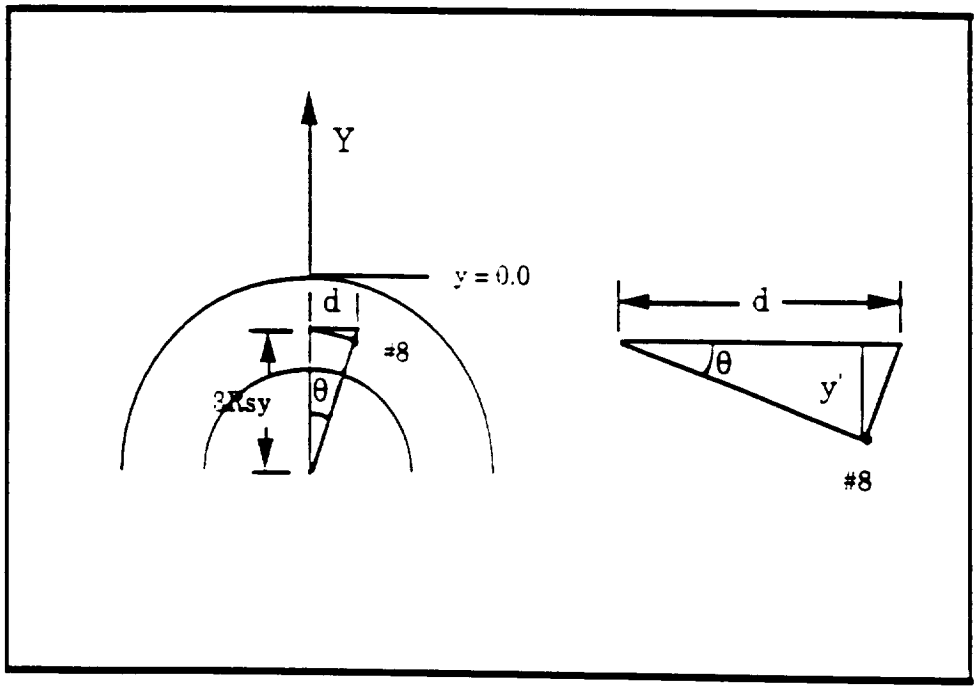


Figure 3.10 Enlarged view of geometry around key point #8.

where

$$\tan(\theta) = \frac{2Rsy}{H'}$$

From the geometric relationship the distance y' can be calculated by

$$y' = d \sin(\theta) \cos(\theta) = d\left(\frac{2Rsy}{(H'^2 + 4Rsy^2)^{1/2}}\right)\left(\frac{H'}{(H'^2 + 4Rsy^2)^{1/2}}\right)$$

or,

$$y' = 3Rsy\left(\frac{2Rsy}{H'}\right)\left(\frac{2Rsy}{(H'^2 + 4Rsy^2)^{1/2}}\right)\left(\frac{H'}{(H'^2 + 4Rsy^2)^{1/2}}\right)$$

$$= \frac{12Rsy^3}{H'^2 + 4Rsy^2}$$

Thus, the y-coordinate of the point #8 is $-Rsy - \frac{12Rsy^3}{H'^2 + 4Rsy^2}$. The space coordinates of the key point are

$$(x,y,z) = \left(\frac{H}{2} + 3Rsy\left(\frac{2Rsy}{H'}\right),\ -Rsy - \frac{12Rsy^3}{H'^2 + 4Rsy^2},\ -2Rsy\right)$$

From Figure 3.9, the coordinates of the key point #9 can readily be determined as:

$$(x,y,z) = \left(\frac{H}{2} + Rsy,\ -Rsy - Rloop,\ 0\right)$$

For the key point #10, where the stitch yarn starts to "wrap" around the off-axis insertion yarn, its geometric relationship with key point#11 and key point#12 is described in Figure 3.11.

In the case of the orientation of insertion yarn is ±45°, the distance A, on the cut cross-section, is equal to $\frac{Rins}{\cos(45°)}$, or, $\sqrt{2}$ Rins. From the observation, the coordinates of the key point#10 can be expressed as:

$$(x,y,z) = \left(\frac{H}{2},\ \frac{H'}{2} - \sqrt{2}\,Rins,\ 7Rins\right)$$

Similiarly, the coordinates of the key point#11 can be found as:

$$(x,y,z) = \left(\frac{H}{2},\ \frac{H'}{2},\ 8Rins + Rsy\right)$$

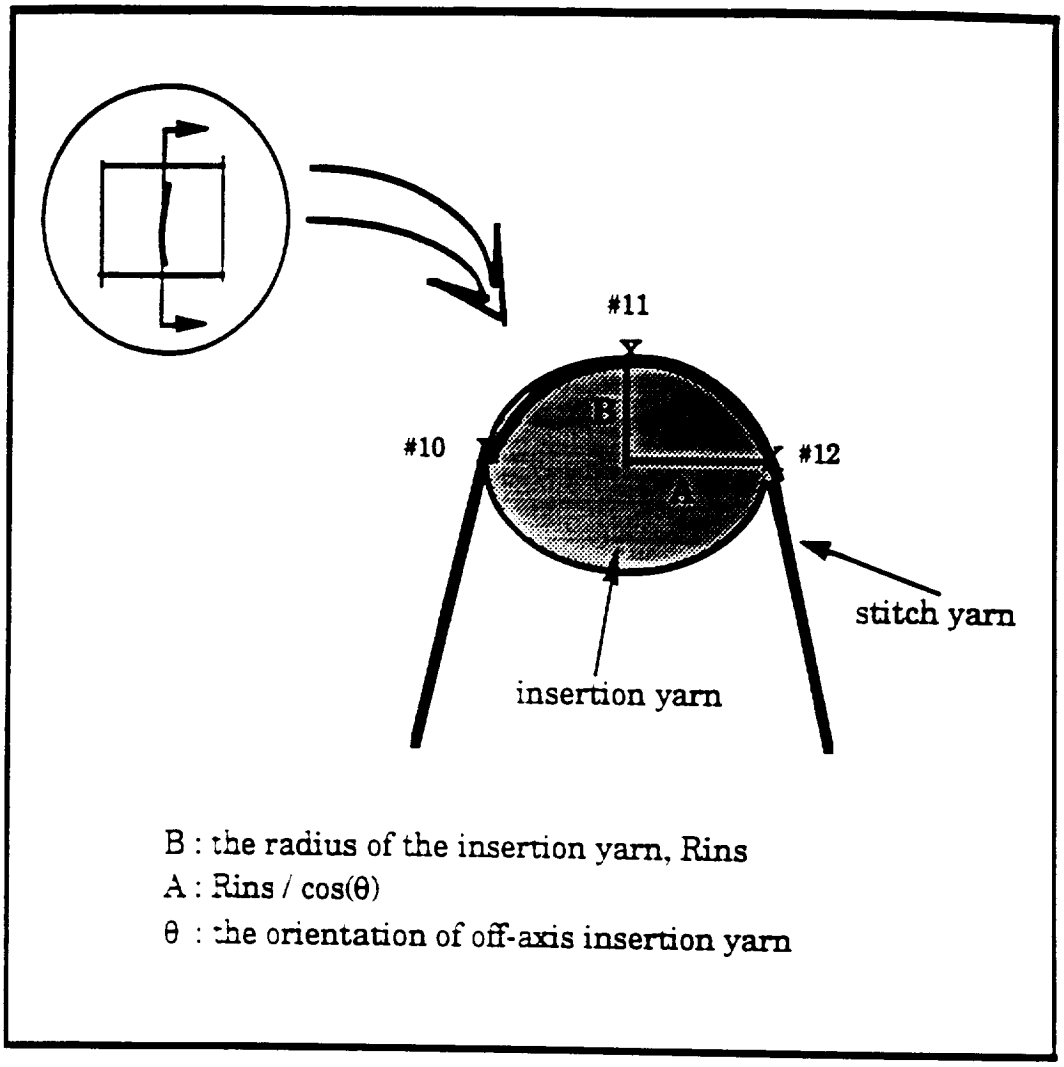


Figure 3.11 Key points in the loop-over portion.

The location of key point#13 is a shift of H' of the key point#9 in y-direction. The coordinates of the key point#13 are

$$(x,y,z) \text{ of the point\#9} + (0,H',0) = (\frac{H}{2} + Rsy,\ H' - Rloop - Rsy,\ 0)$$

Similarly, the location of key point#14 is a shift of H' of the key point#8 in y-direction. The coordinates of the key point#14 are

$$(x,y,z) \text{ of the point\#8} + (0,H',0)$$

$$= (\frac{H}{2} + 3Rsy(\frac{2Rsy}{H'}),\ H' - Rsy - \frac{12Rsy^3}{H'^2 + 4Rsy^2},\ -2Rsy)$$

For the key points from #15 to #19, their locations are a shift of H' in y-direction to the key points from #7 to #3 (in reverse order), respectively. The key points #20 and #14 are symmetrical with respect to the line : $x = \frac{H}{2}$. The x-coordinate of key point #20 can be found by:

$$\frac{H}{2} - (\text{x-coordinate of the key point \#14} - \frac{H}{2})$$

$$= H - (\text{x-coordinate of the key point \#14})$$

$$= \frac{H}{2} - 3Rsy\,(\frac{2Rsy}{H'})$$

Thus, the coordinates of the key point #20 are expressed by:

$$(\frac{H}{2} - 3Rsy\,(\frac{2Rsy}{H'}),\ H' - Rsy - \frac{12Rsy^3}{H'^2 + 4Rsy^2},\ -2Rsy)$$

In the same sense, the key points #21 and #13 are symmetrical with respect to the line : $x = \frac{H}{2}$. The x-coordinate of key point #21 can be found by:

$$\frac{H}{2} - (\text{x-coordinate of the key point \#13} - \frac{H}{2})$$

$$= H - (\text{x-coordinate of the key point \#13})$$

$$= \frac{H}{2} - Rsy$$

The coordinates of the key point #21 are given by:

$$(\frac{H}{2} - Rsy,\ H' - Rloop - Rsy,\ 0)$$

The locations of the key points #22,#23,#24, #25 and #26 are a shift of H' in y-direction to the key points #10, #11, #12, #21 and #20, respectively. The location of the key point #27 is a shift of 2H' in y-direction to the key points #2, and the coordinates of the key point #27 are

$$(x, y, z) = (x, y, z) \text{ of the key point \#2} + (0, 2H', 0)$$

$$= (\frac{H}{2} - Rsy - 4Rsy(\frac{2Rsy}{H'}),\ 2H',\ -2Rsy + \frac{Rsy}{H' - 3Rsy})$$

The results of the geometric analysis for the stitch yarn will be used as input for solid modeling of the MWK. The summary of the coordinates of each key point is shown in Table 3-2.

key point	x - coordinate	y - coordinate	z - coordinate
1	$\frac{H}{2}$ - Rsy	- Rsy - Rloop	0
2	$\frac{H}{2}$ - Rsy - 4Rsy($\frac{2Rsy}{H'}$)	0	- 2Rsy + $\frac{Rsy}{H' - 3Rsy}$
3	$\frac{H}{2}$ - Rloop	H' - Rsy - Rloop	0
4	$\frac{H}{2}$ - $\frac{3}{2}\sqrt{2}$ Rsy	H' -Rsy -Rloop(1- $\frac{\sqrt{2}}{2}$)	0
5	$\frac{H}{2}$	H' - Rsy	0
6	$\frac{H}{2}+\frac{3}{2}\sqrt{2}$ Rsy	H' -Rsy -Rloop(1- $\frac{\sqrt{2}}{2}$)	0
7	$\frac{H}{2}$ - Rloop	H' - Rsy - Rloop	0
8	$\frac{H}{2}$ + 3Rsy ($\frac{2Rsy}{H'}$)	-Rsy - $\frac{12Rsy^3}{H'^2 + 4Rsy^2}$	-2Rsy
9	$\frac{H}{2}$ + Rsy	- Rsy - Rloop	0
10	$\frac{H}{2}$	$\frac{H'}{2}$ - $\sqrt{2}$ Rins	7Rins
11	$\frac{H}{2}$	$\frac{H'}{2}$	8Rins + Rsy
12	$\frac{H}{2}$	$\frac{H'}{2}$ + $\sqrt{2}$ Rins	7Rins
13	$\frac{H}{2}$ + Rsy	H' - Rsy - Rloop	0
14	$\frac{H}{2}$ + 3Rsy ($\frac{2Rsy}{H'}$)	H'-Rsy- $\frac{12Rsy^3}{H'^2 + 4Rsy^2}$	-2Rsy
15	$\frac{H}{2}$ - Rloop	2H' - Rsy - Rloop	0
16	$\frac{H}{2}+\frac{3}{2}\sqrt{2}$ Rsy	2H' -Rsy -Rloop(1- $\frac{\sqrt{2}}{2}$)	0
17	$\frac{H}{2}$	2H' - Rsy	0
18	$\frac{H}{2}$ - $\frac{3}{2}\sqrt{2}$ Rsy	2H' -Rsy -Rloop(1- $\frac{\sqrt{2}}{2}$)	0

19	$\frac{H}{2}$ - Rloop	2H' - Rsy - Rloop	0
20	$\frac{H}{2}$ - 3Rsy ($\frac{2Rsy}{H'}$)	H' - Rsy - $\frac{12Rsy^3}{H'^2 + 4Rsy^2}$	-2Rsy
21	$\frac{H}{2}$ + Rsy	H' - Rsy - Rloop	0
22	$\frac{H}{2}$	$\frac{3H'}{2}$ - $\sqrt{2}$ Rins	7Rins
23	$\frac{H}{2}$	$\frac{3H'}{2}$	8Rins + Rsy
24	$\frac{H}{2}$	$\frac{3H'}{2}$ + $\sqrt{2}$ Rins	7Rins
25	$\frac{H}{2}$ + Rsy	2H' - Rsy - Rloop	0
26	$\frac{H}{2}$ - 3Rsy ($\frac{2Rsy}{H'}$)	2H' - Rsy - $\frac{12Rsy^3}{H'^2 + 4Rsy^2}$	-2Rsy
27	$\frac{H}{2}$ - Rsy - 4Rsy($\frac{2Rsy}{H'}$)	2H'	- 2Rsy + $\frac{Rsy}{H' - 3Rsy}$

3.4 References

3.1 Pastore, C. M. and Cai, Y. J., "Application of computer aided geometric modeling for textile structural composites", Composite Materials Design and Analysis, ed. W. P. de Wilde and W. R. Blain, pp. 127 - 141, Computational Mechanics Publications, Springer-Verlag, 1990.

3.2 de Boor, C., A Practical Guide to Splines, Springer Verlag, New York, 1979.

Chapter - 4 Visualization Verification

Based on the techniques described in Chapter 2 and Chapter 3, the preform geometries of 3-D braid and MWK can be generated by computer simulation program. In order to verify the simulated geometries, the microsturctures of real composites were examined by photo microscopy. The composites were cut, mounted with resin, polished using polishing powder. Pictures were taken after each polish and can be used to compare with simulated graphics.

4.1 Material System

4.1.1 3-D Braid

Both 3-D braided PEEK/graphite and Epoxy/Milliken multiaxial warp knit composites were used in the present study. The 3-D braided preforms were made by using commingled Victrex® 150G PEEK/AS4 6K yarn manufactured by BASF Structural Materials, Inc. The commingled yarn contains approximately 280 PEEK fibers (fiber diameter ~ 27 micrometers) and 6,000 graphite fibers (fiber diameter ~ 8 micrometers). The graphite fiber volume fraction in each commingled yarn is 62.7%. In order to obtain a composite with corss section of 0.254 mm (0.1 inch) by 1.27 mm (0.5 inch), with a braiding angle of ±20°, a 1x1 construction was produced by using 72 yarns in a loom consisting of 18 columns by 4 tracks. After braiding, the 3-D braided preforms were placed in a carbon steel mold and consolidated by using a hot press, and held at 400°C and 1.38 MPa (200psi) for an hour. After that, the mold was taken out from the hot press and placed in a cold press and held at 1.38 MPa for another 30 minutes.

4.1.2 Multiaxial Warp Knit

The Epoxy/Milliken multiaxial warp knit composite used in this study was provided by NASA. The fiber architecture design of this material is shown in Figure 4.1. It was fabricated with Hercules AS-4 epoxy-sized carbon tows. The 0° and 90° plies were produced with 12 K tows, whereas 9K tows were used for the ±45° plies. This tow distribution was required to achieve similar fiber area for each ply because of the different tow count for the off-axis plies.

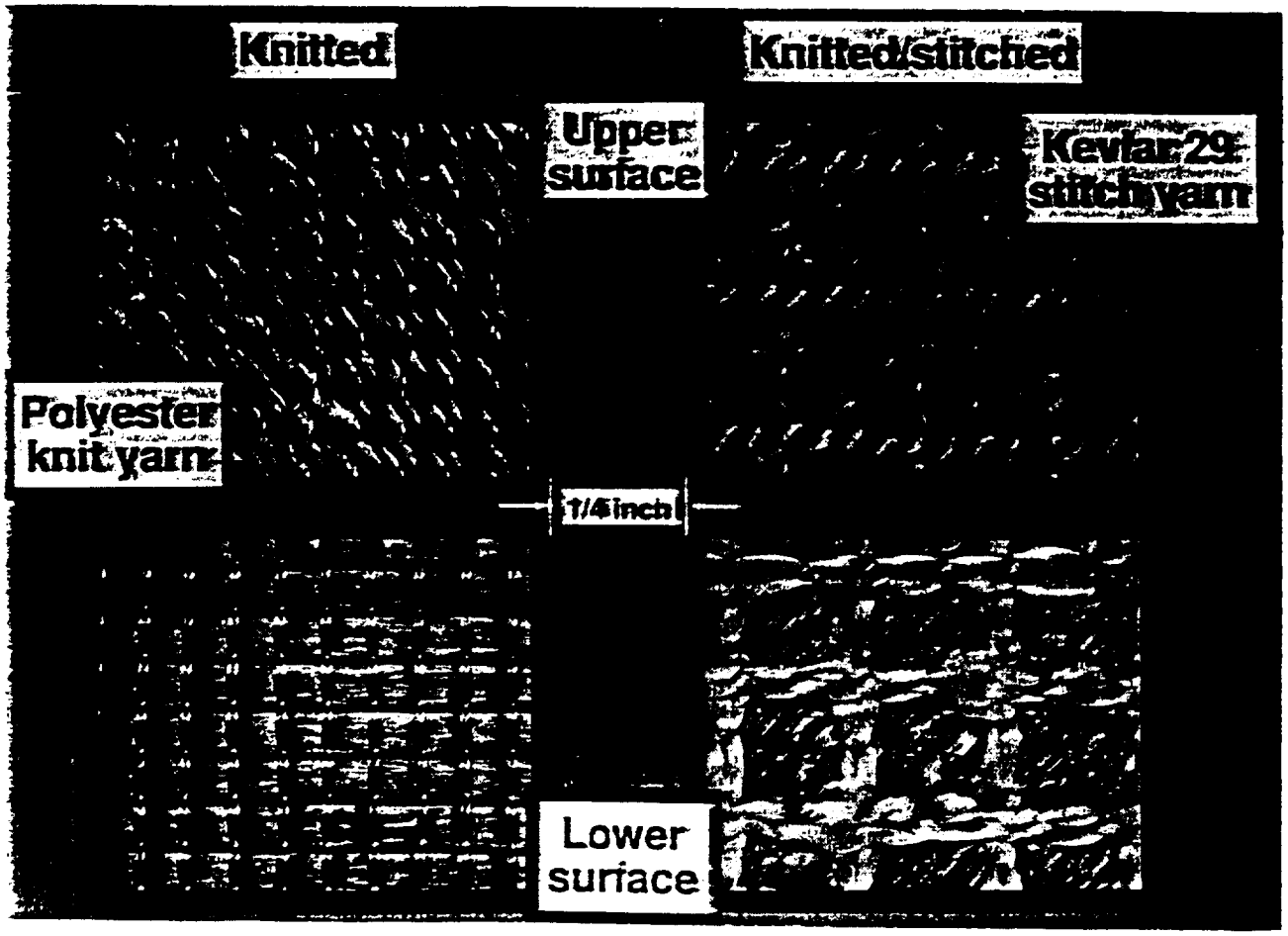


Figure 4.1 Fabric design of the multiaxial warp knit composites.

Only the 4-ply subgroup (-45°, +45°, 0°, 90°) shown in Figure 4.2 was included for this investigation. An additional 4-ply subgroup (+45°, -45°, 0°, 90°) would be required to produce a symmetric quasi-isotropic fabric preform. Four 4-ply subgroups were stacked to form the full 16-ply laminate. Note that the 16-ply stack, shown in Figure 4.2, is unsymmetric. The 4-ply subgroup was produced by chain-knitting the carbon tows together with a polyester yarn. Some of the 16-ply stacks were stitch together with kevlar-29 1500 denier stitching yarn using a chain stitch.

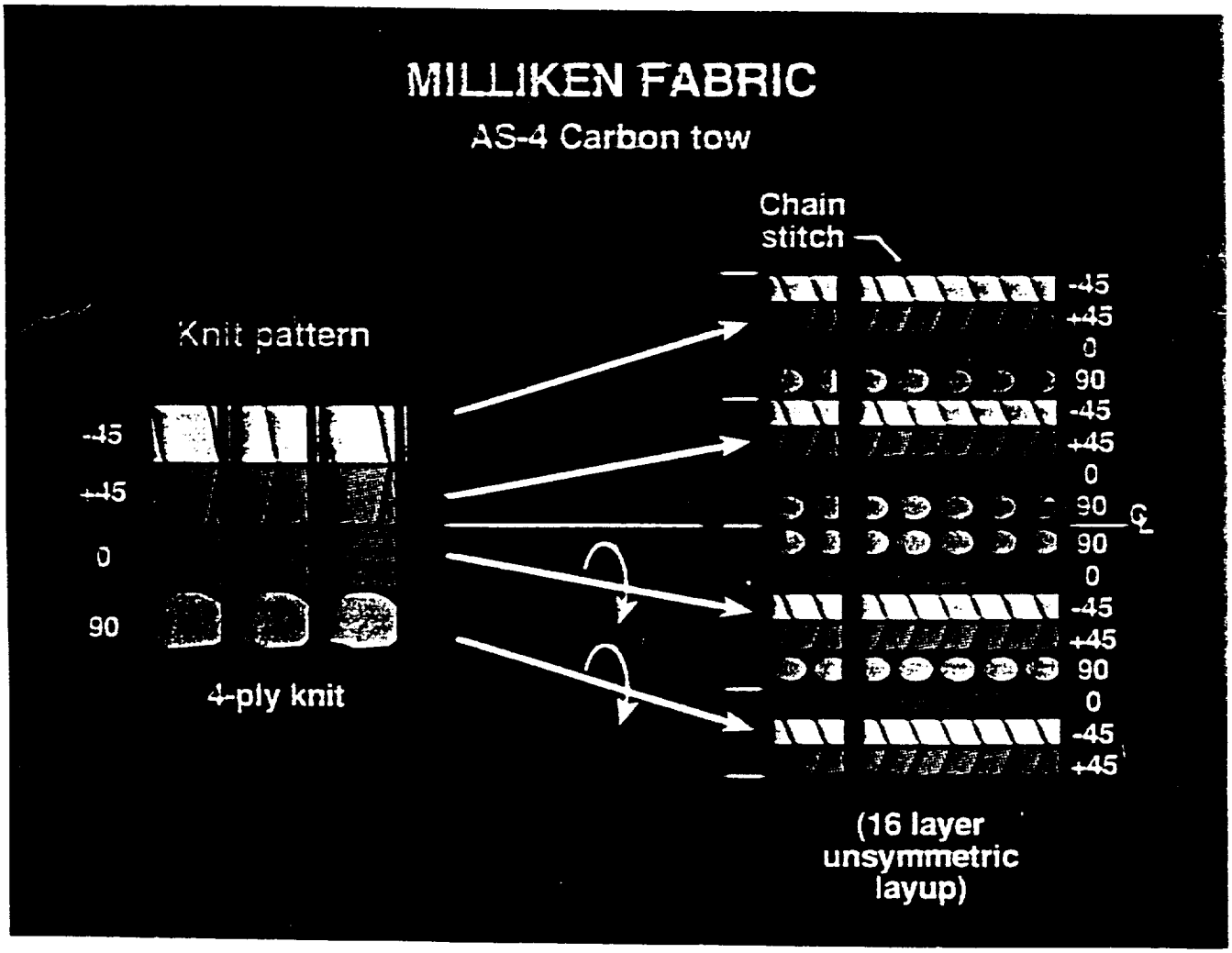


Figure 4.2 Stacking seguence of the 16 plied-MWK results in an unsymmetric conformation.

4.1.3 Sample Preparation

The unit cell geometries of 3-D braided and multiaxial warp knit composites were observed by the optical microscope technique. In this study, the cross-section of the 3-D braided composite was cut perpendicular to the braiding axis, whereas the 0°-direction and 90° orientation cross sections of the multiaxial warp knit composites were cut for examination, as illustrated in Figure 4.3.

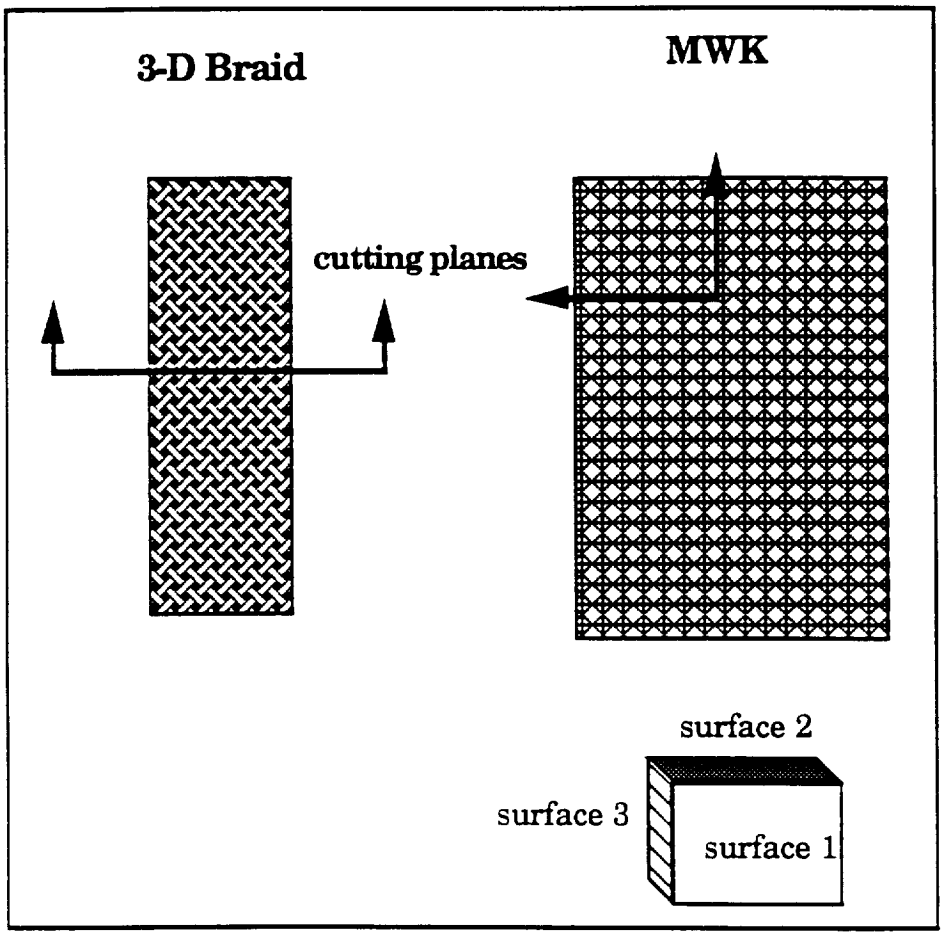


Figure 4.3 Schematic of sectioning

The cut-offs were mounted in a holder with Lecoset® 7000 cold-curing resin manufactured by Leco corporation. After the resin is cured, the samples were polished with 320, 400, 600 grid silicon carbide polishing paper , as well as 15, 5 micron aluminum oxide powder and 1 micron alpha alumina powder, to obtain smooth surfaces for photos. All polished samples were examined under an optical microscope with 32x magnification factor. The montages are expected to show the variety of the unit cell structure of these samples from

different cutting planes. Therefore, in order to study the variation of the unit cell structures in 3-D space, a series of montage of the unit cell structures of these composites were generated by progressive sectioning of the above samples. By doing so, the 3-D fiber conformation of the composite can be recorded on a series of planes with certain distance apart and thus, the 3-D fiber architecture can be envisioned and described.

4.2 3-D Braid

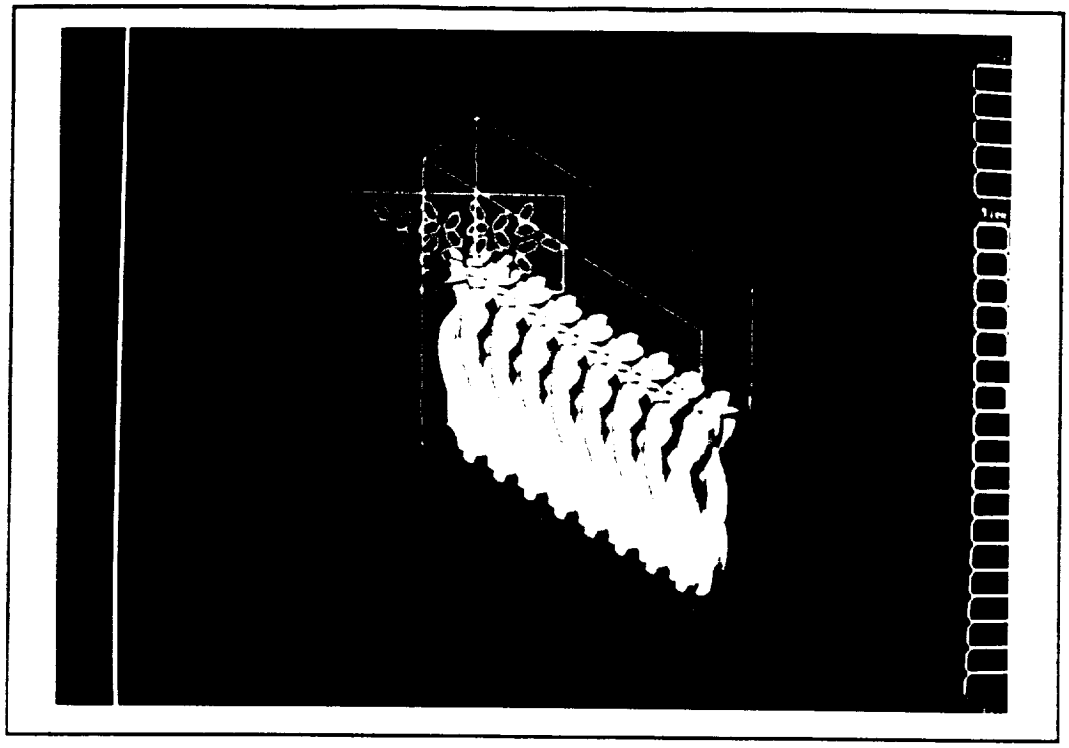
4.2.1 Computer Generated Graphics

The analysis of textile composites depends directly on fiber architecture of the composites, that, in turn, can be accurately characterized by a computer aided geometric models. The geometric models for the 3-D braided preforms are given in Chapter Three, in which the geometric model considers the relative motions of the tracks and columns in the braiding machine and generates a mathematical simulation of the braiding process.

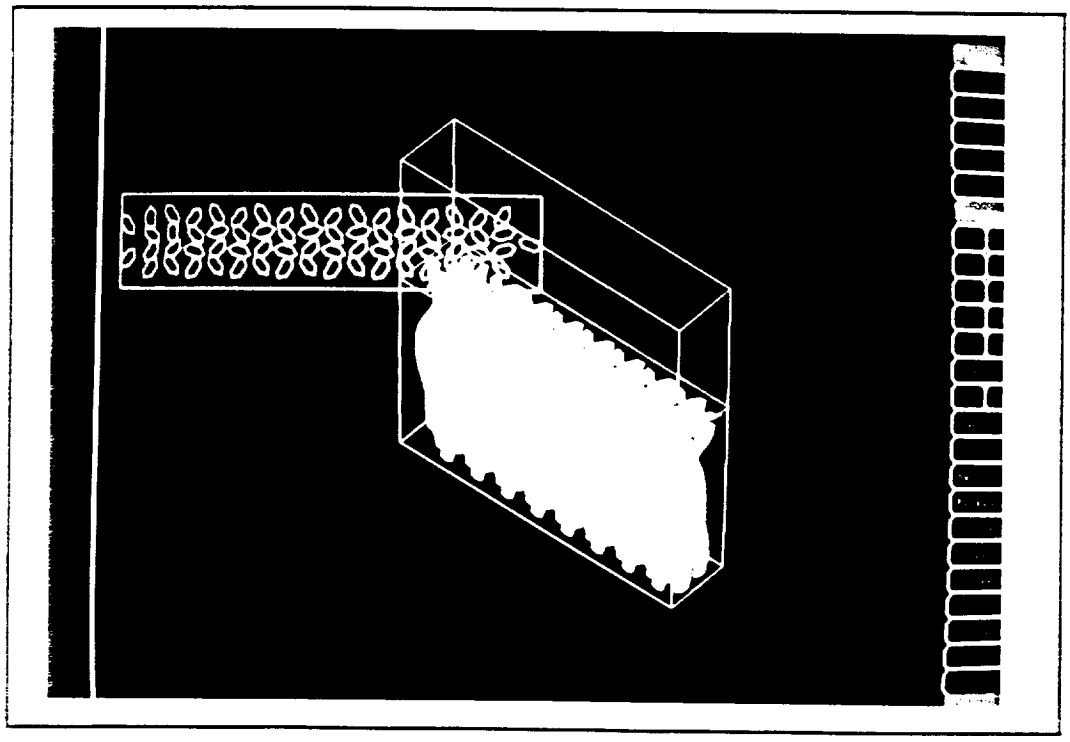
In order to compare the computer-generated graphics with experimental observations, a virtual loom of 4 tracks and 18 columns is set up on the computer. Six track/column movements are simulated, which will construct 1.5 unit cells in the braiding axis. The yarn cross-section is assumed to be ellipse with the ratio of major axis over minor axis to be 0.5. The graphics are generated from a Sun workstation. The details of the implementation of the graphics model can be found in Chapter 6.

Figure 4.4(a) - (h) show a series of cut cross-sections of a 3-D braid. The graphs cover a track/column movement cycle. The loom state can be seen from each cut cross-section. First, the columns move up/down, as shown from Figure 4.4(a) to (c). In the next step, the tracks travel back/forth horizontally, as shown from Figure 4.4(d) to (f). Figure 4.4(g) and Figure 4.4(h) show the loom state after column movement and track movement, respectively. Therefore, the braiding pattern on each cut plane represents a loom state. As we go on examining the braiding pattern along the braiding axis, the braiding pattern will repeat after a unit cell's length. The braiding pattern generated from computer is slightly different from the experimental observations, especially around the boundary. This is due to the compression while the composite was consolidated by hot pressing.

As observed from Figure 4.4(a) to (h), the braiding pattern changes 90° between Figure 4.4 (a) and (b). The braiding pattern changes 180° between Figure 4.4(c) and (g). Thus,

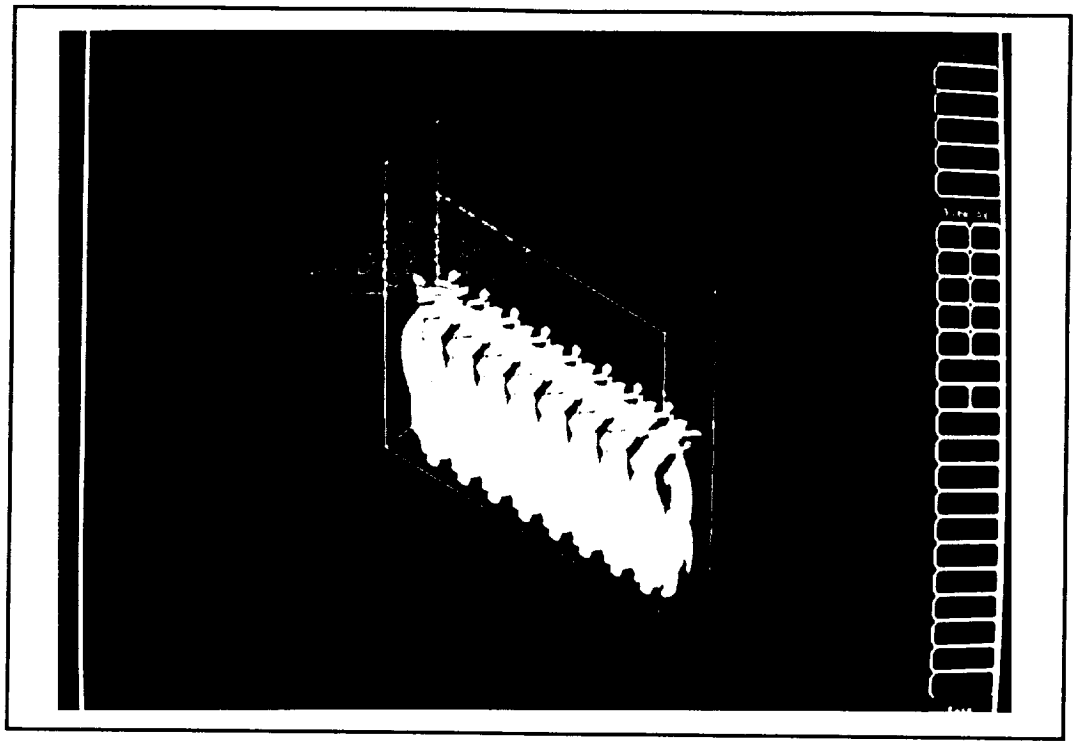


(a)

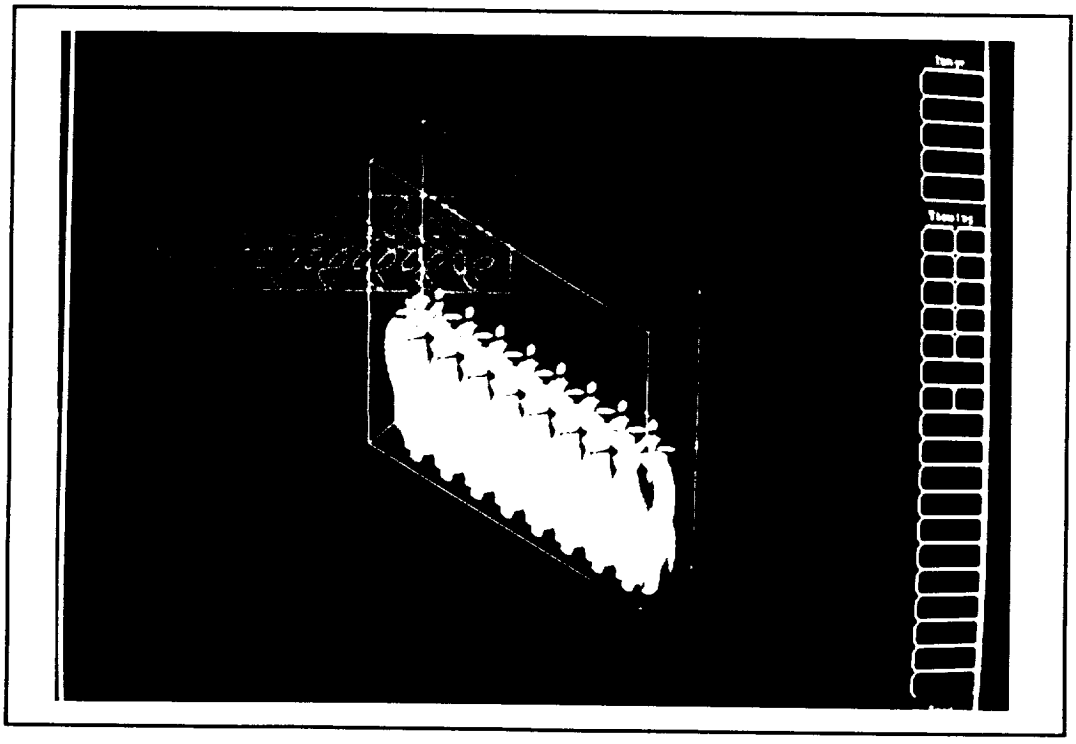


(b)

Figure 4.4 Cut sections of a 3-D braided composite.

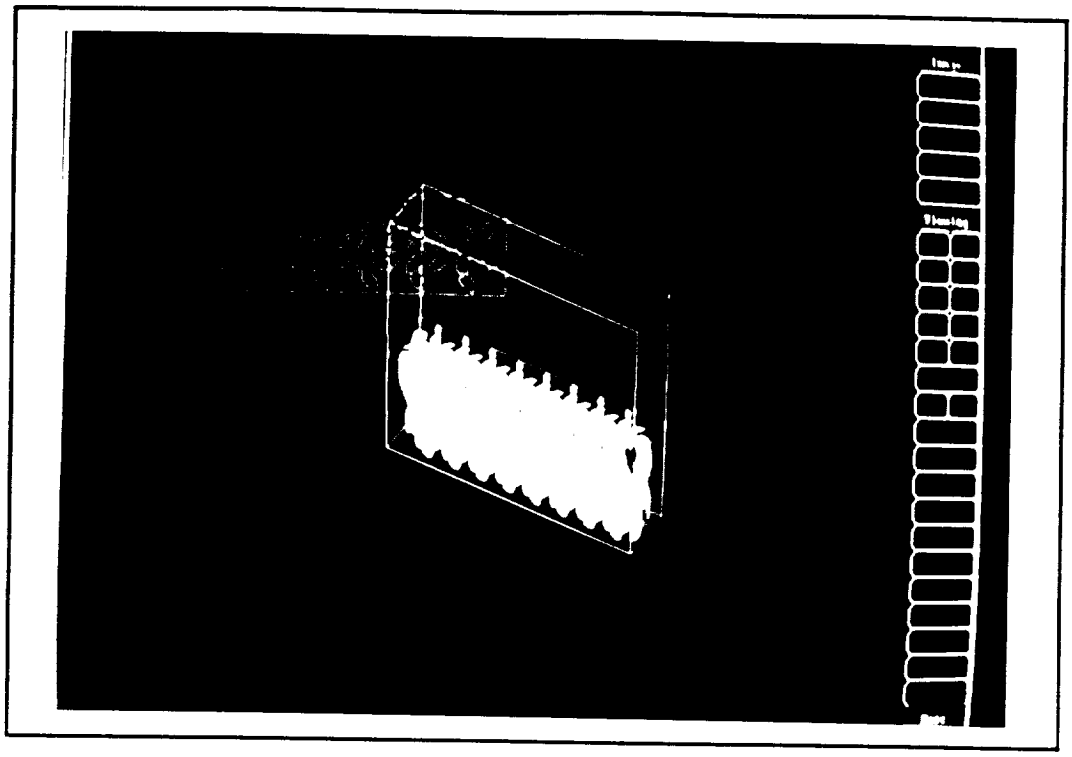


(c)

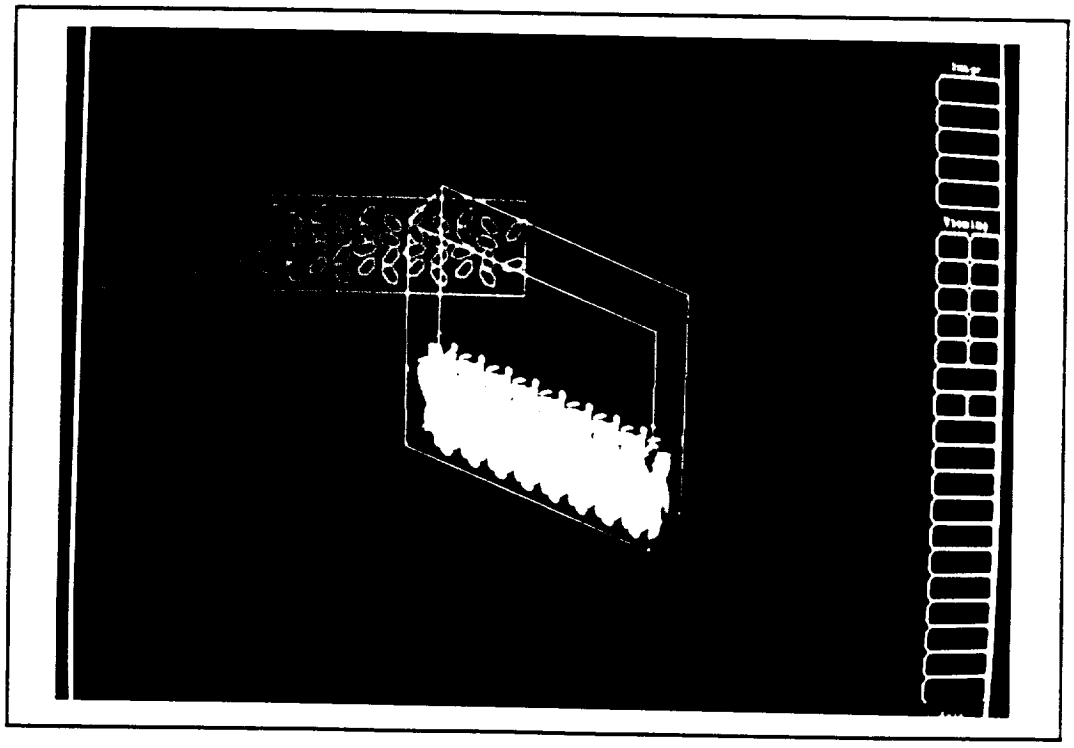


(d)

Figure 4.4 Continued.

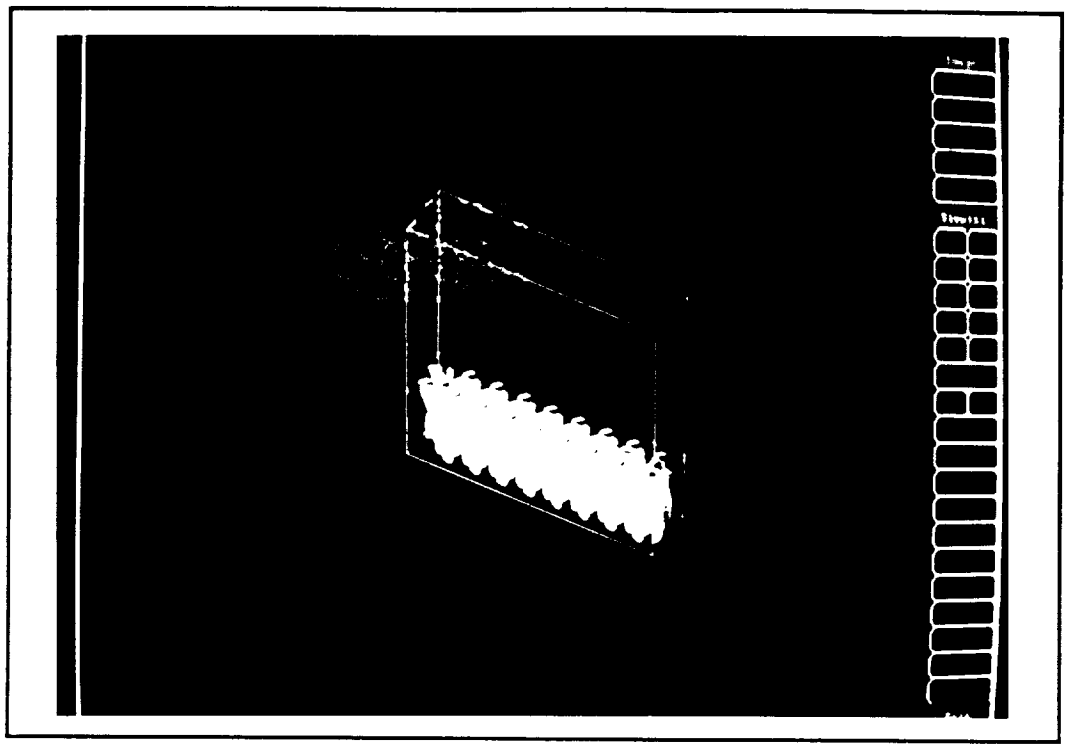


(e)

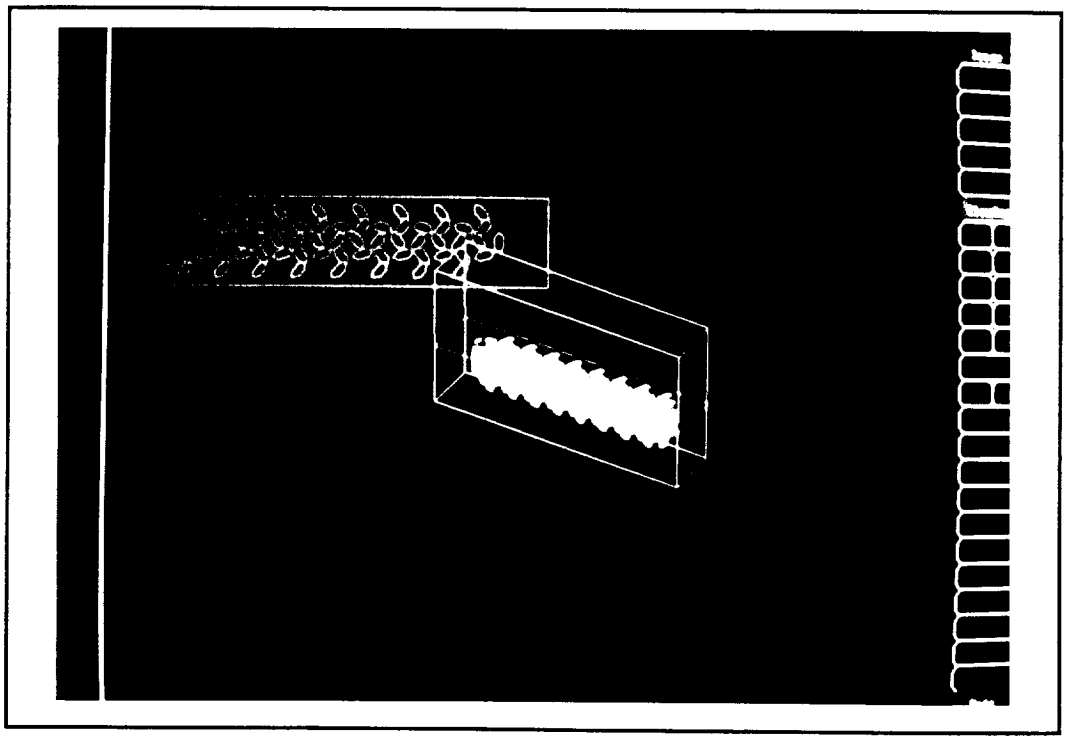


(f)

Figure 4.4 Continred.



(g)



(h)

Figure 4.4 Continued.

if the cut planes are the same as the planes on which the photo pictures are taken, the computer-generated graphs should resemble the pictures in the central portion of the specimen.

4.2.2 Experimental Verification

The unit cell structure of 3-D braided PEEK/graphite composite is a diamond-shape structure. The diagonal along the braiding axis was measured at 5.1 mm. The schematic of the planes which pictures were taken is shown in Figure 4.5. The distances between each planes are 0.2mm, 1.6mm and 2.4mm, respectively, which is long enough to represent the variation of the structure within a unit cell. The montages showing the cross-section of the composite along the braiding axis in these four different layers are shown in Figures 4.6. As it can be seen in the figure, the fiber packing pattern changes along the braiding axis. The first two pictures, taken from plane A and plane B, respectively, show no significant difference because of the short distance separating the planes. The 90° fiber packing pattern change between plane B and plane C suggests that the unit cell consists of at least two pairs of yarns. These two pairs of yarns are othogonal to each other. The two yarns in each pair are interlaced each other. The distance between plane C and plane D is about half length of a unit cell. The fiber pattern on plane C is about 180° to the fiber pattern on plane D. By induction, it can be predicted that two montages will look the same if they are viewed on the planes 5.1 mm apart.

In conclusion, the unit cell geometry of 3-D braided composites can be described and visualized by the computer software.

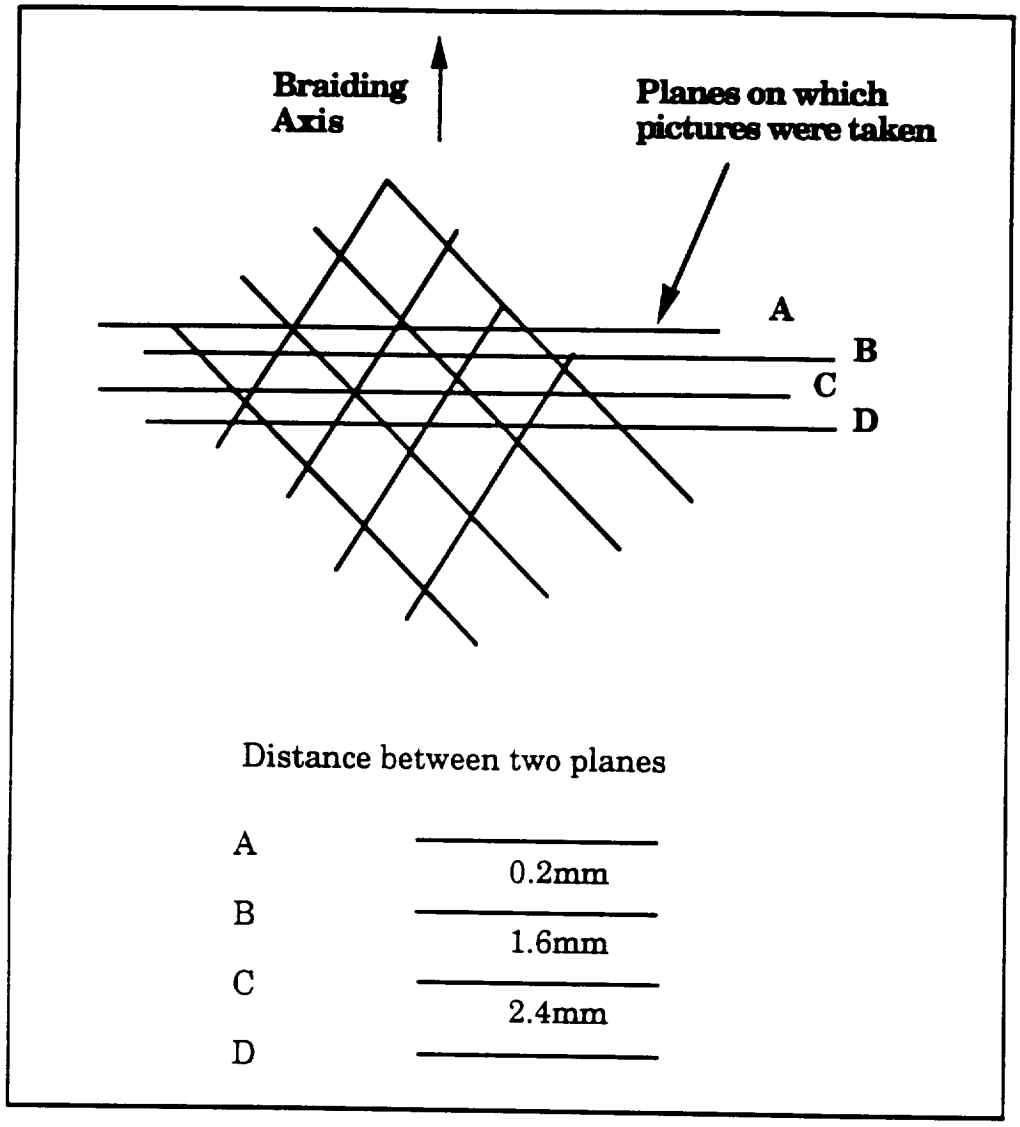


Figure4.5 Schematic of planes on which pictures are taken for 3-D braid.

(a) A - plane

(b) B - plane

(c) C - plane

(d) D - plane

Figure 4.6 Photomicrograph of 3-D braided composites.

4.3 Multiaxial Warp Knit

4.3.1 Computer Generated Geometry

The geometric model for the MWK preform is given in Chapter Three, in which the geometric model relates the key parameter of fiber architecture to the processing variables base on the idealized cross-sections of insertion yarns and the path of stitch loops. Combined with the optical observations detailed in Section 4.3.2, the geometric model can be translated into computer solid models for MWK preforms, which facilitates the visualization of detailed internal geometries for MWK preforms and the identification of their unit cells.

In order to obtain an overview of the figures from simulation, a schematic of the planes on which the computer generated grapgs were taken is shown in Figure 4.7. In the figure, a unit cell, which represents the geometry of the whole structure, is used to illustrate the planes. Later, the computer code will generate the corresponding graphs based on the planes shown.

Based on the geometric modeling and computer graphics techniques, the MWK preform with 8 unit cells is generated on Sun workstation. Figure 4.8 shows the top view, side view, back view and (1 1 1) view of the MWK preform, respectively. As can be seen, the stitch yarn loops over insertion yarns with a chain configuration. The geometry of the stitch yarn is simulated according to the geometric model, described in Chapter two and three. The actual geometry of the stitch yarn can be determined by the yarn tension itself, yarn friction coefficient and yarn bending rigidity. However, this is not in the scope of this project. Figure 4.9 shows the (1 0 0) cuting plane and cutting section in top view and side view, respectively. From a little distance to the previous cutting plane, a section is cut , as shown in Figure 4.10. The cutting planes are corresponding to the polishing planes, described in the next section. Since the cutting plane does not cut through 0° insertion yarn, the 90° yarn does not show up in the cut section.

Figure 4.11 shows the (0 1 0) cuting plane and cutting section in top view and side view, respectively. From a little distance to the previous cutting plane, a section is cut , as shown in Figure 4.12. Figure 4.13 shows the (0 0 1) cuting plane and cutting section in top view and side view, respectively. From a little distance to the previous cutting plane, a section is cut , as shown in Figure 4.14. The cutting planes are chosen according to polishing planes. The computer generated graphics will be compared with the photomicrographs from the polished sample.

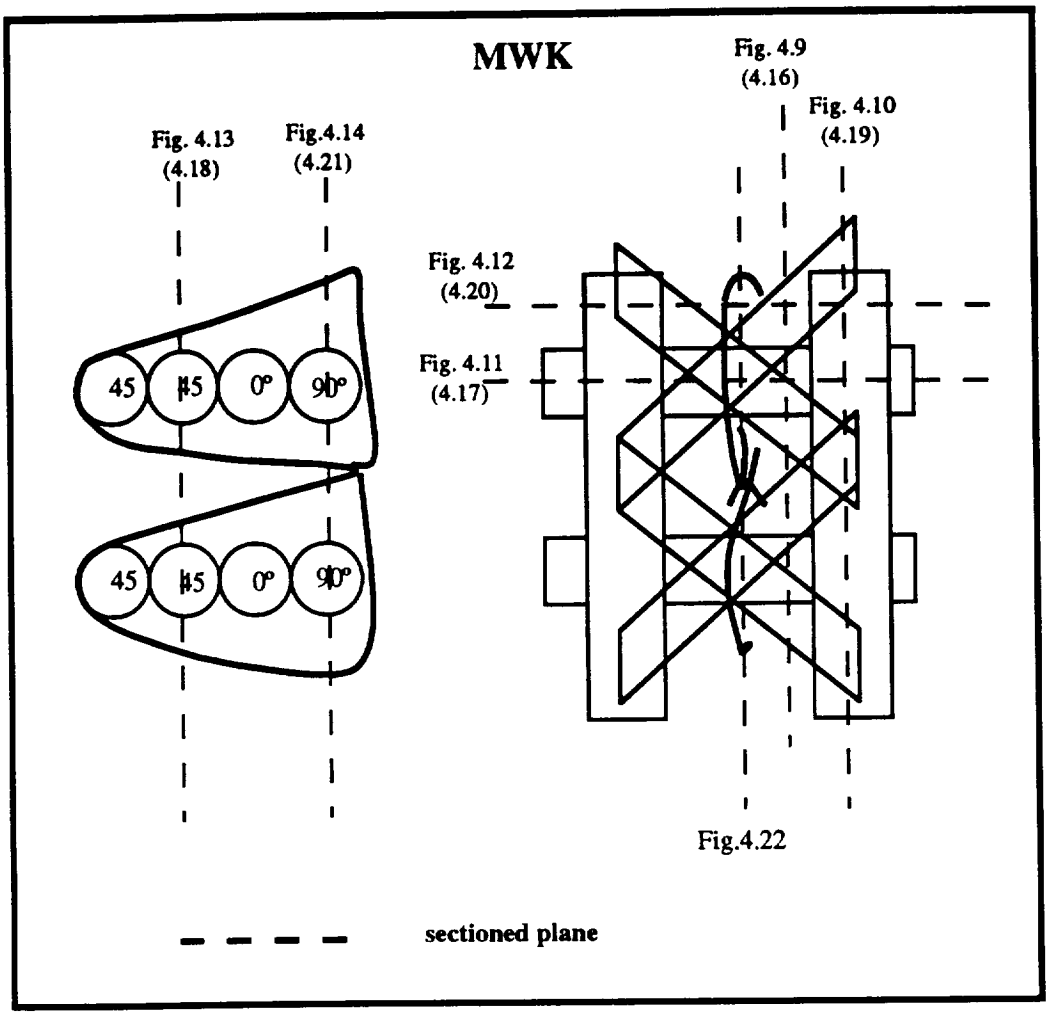
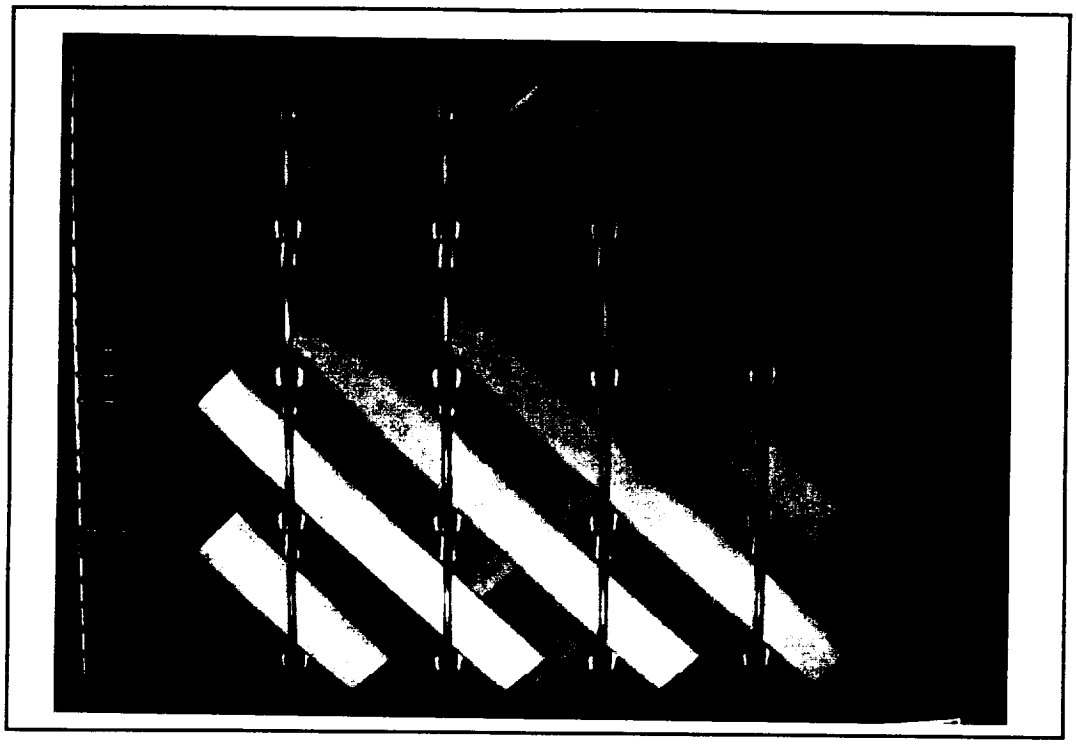
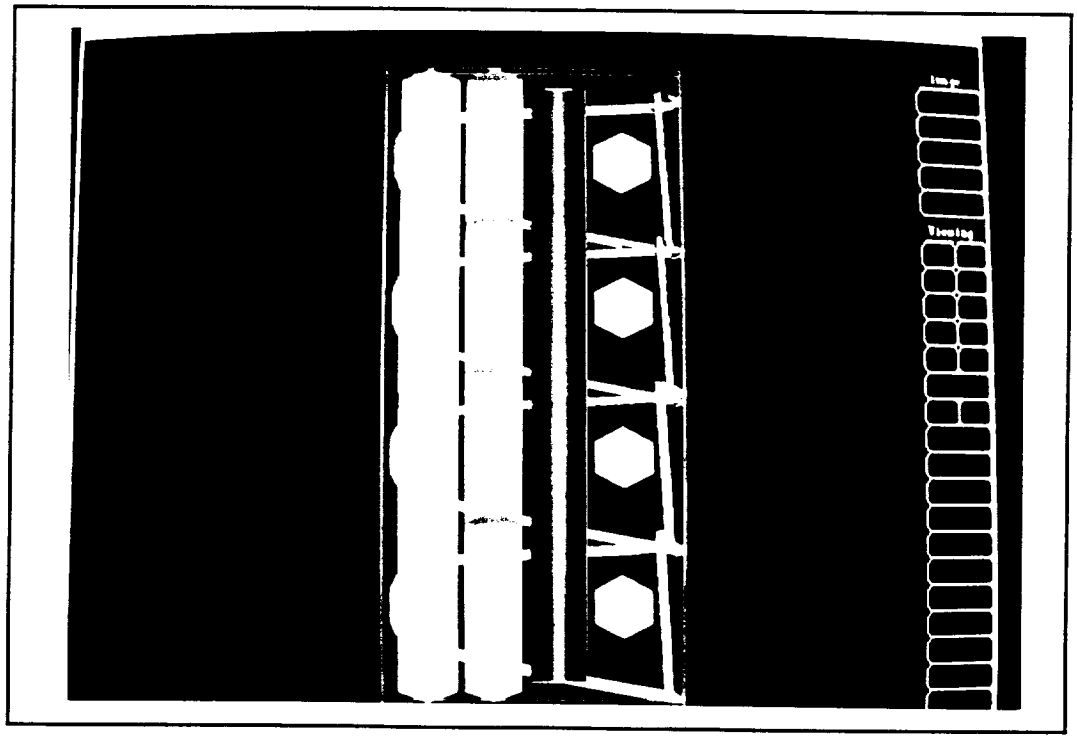


Figure 4.7 Schematic of simulated planes.

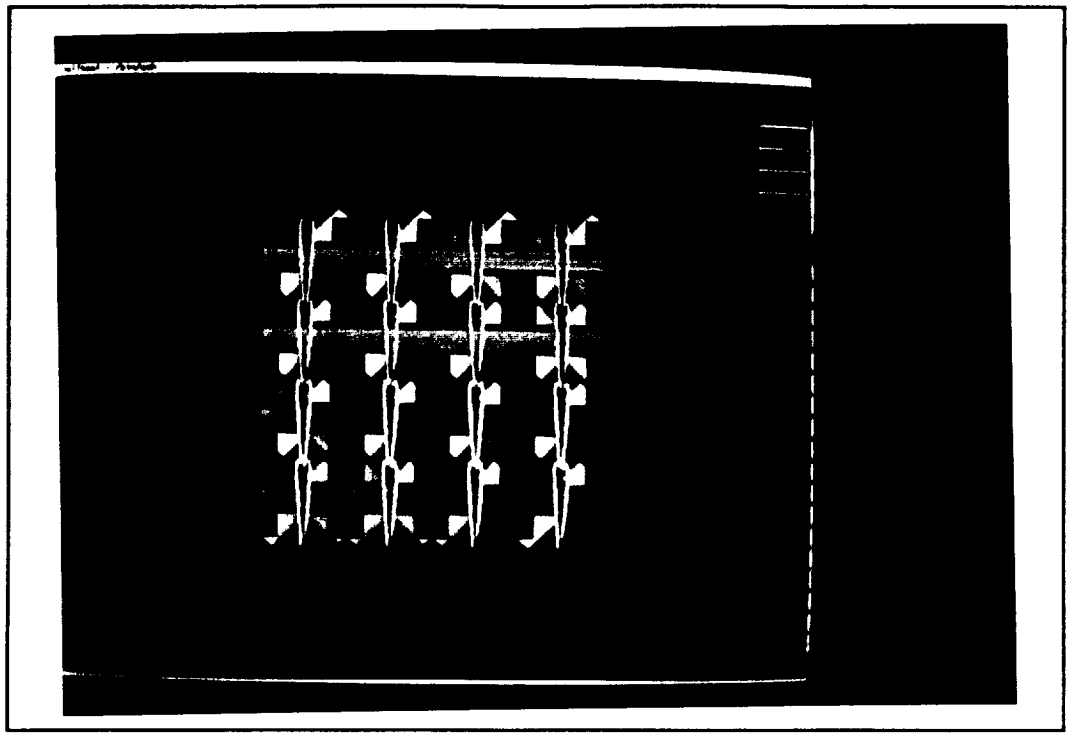


(a) top view

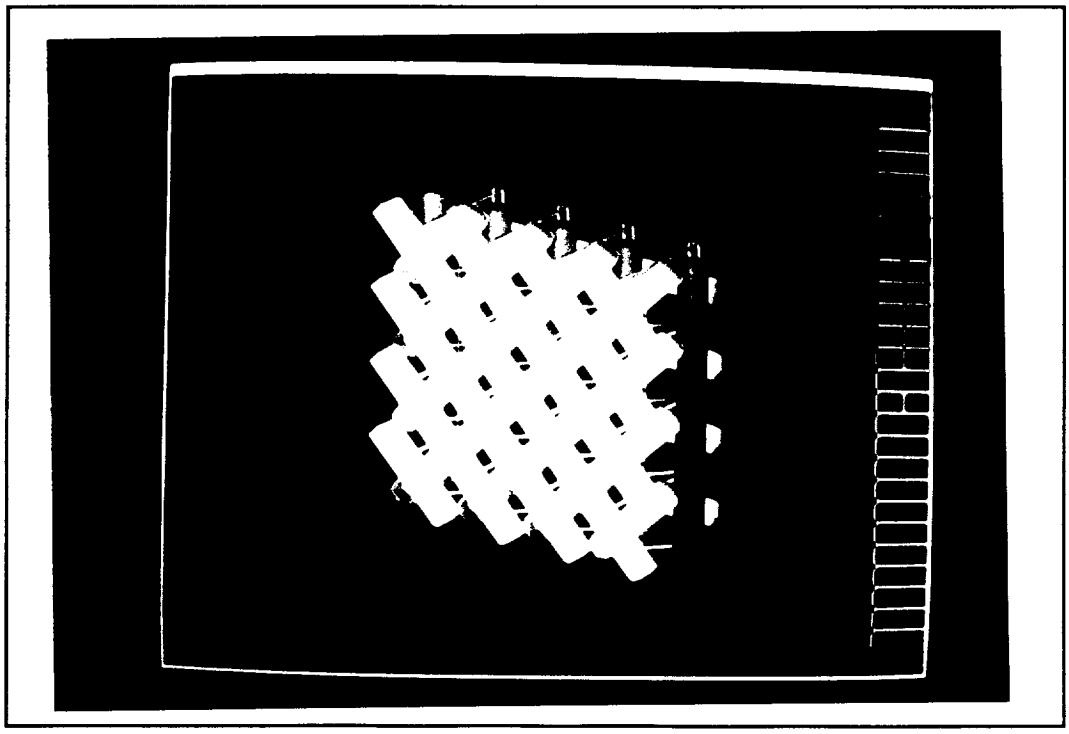


(b) side view

Figure 4.8 Computer rendering of a MWK preform.

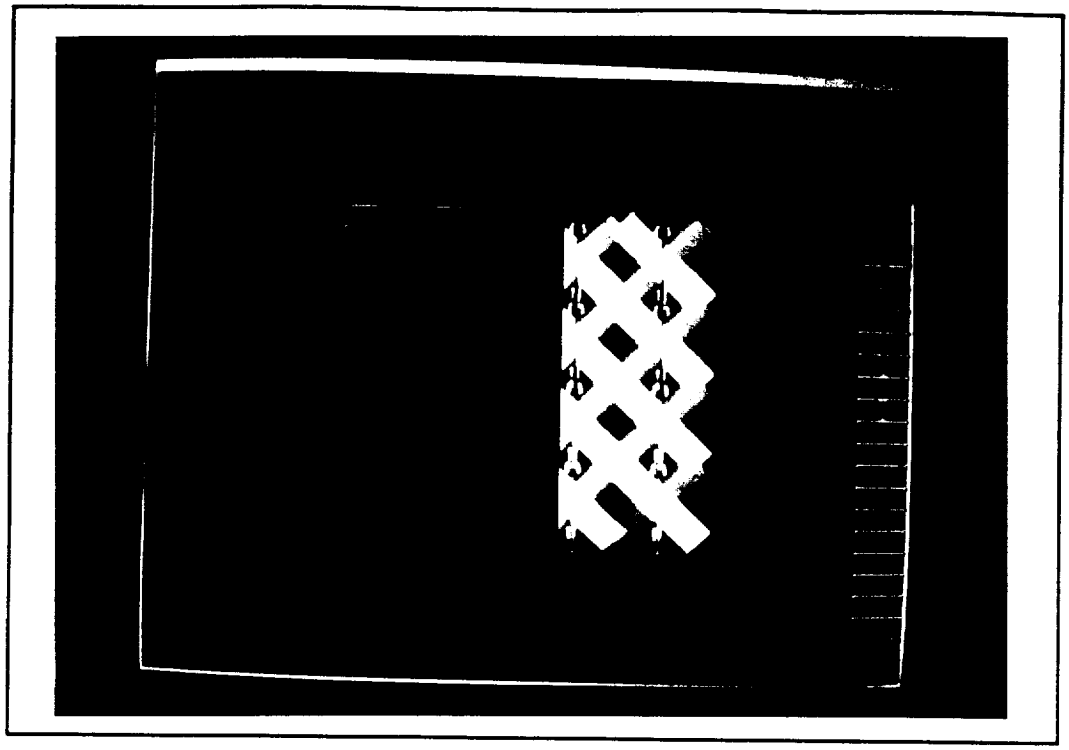


(c) back view

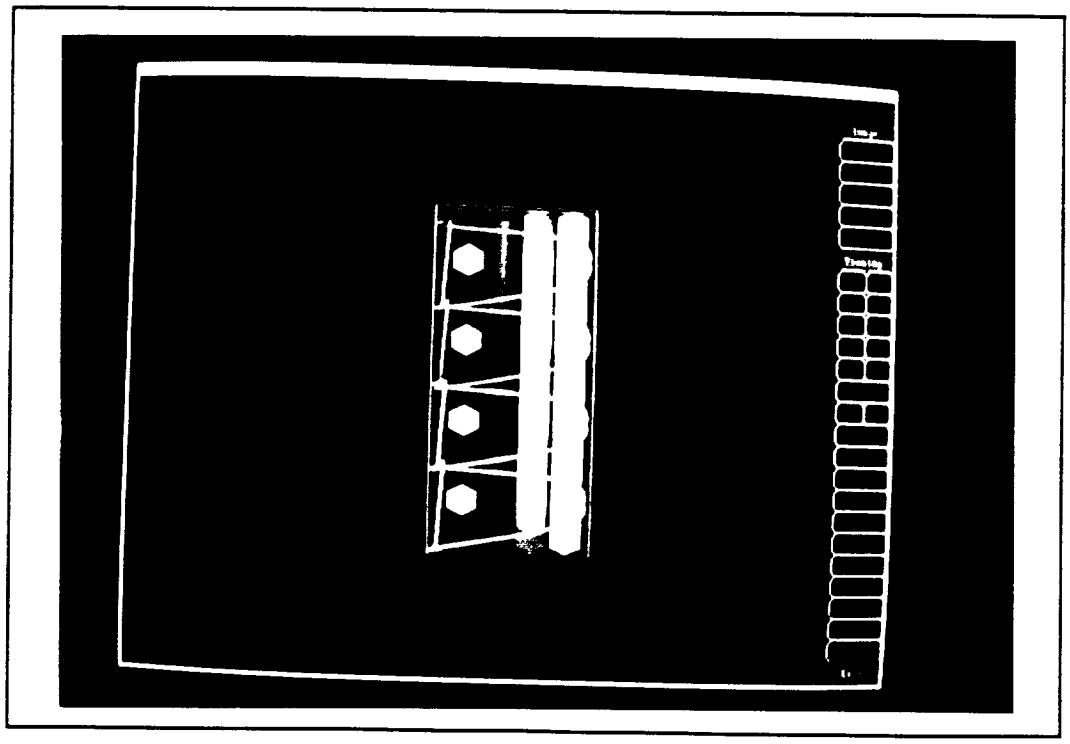


(d) (1 1 1) view

Figure 4.8 Continued.

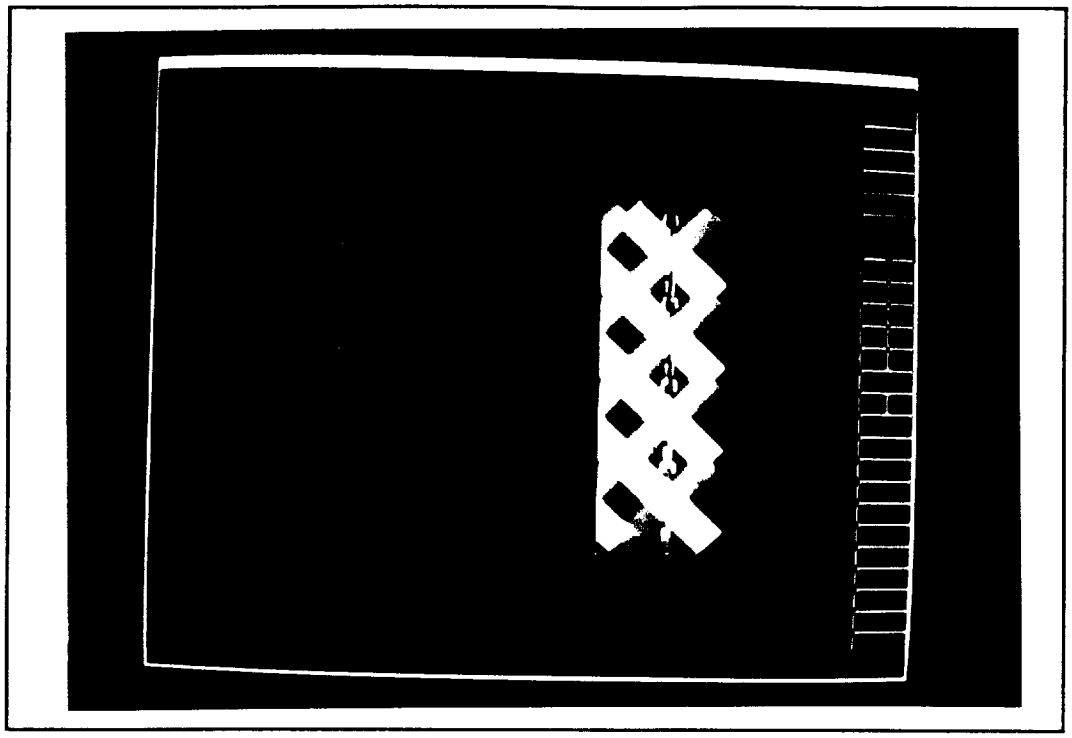


(a) top view and cut section

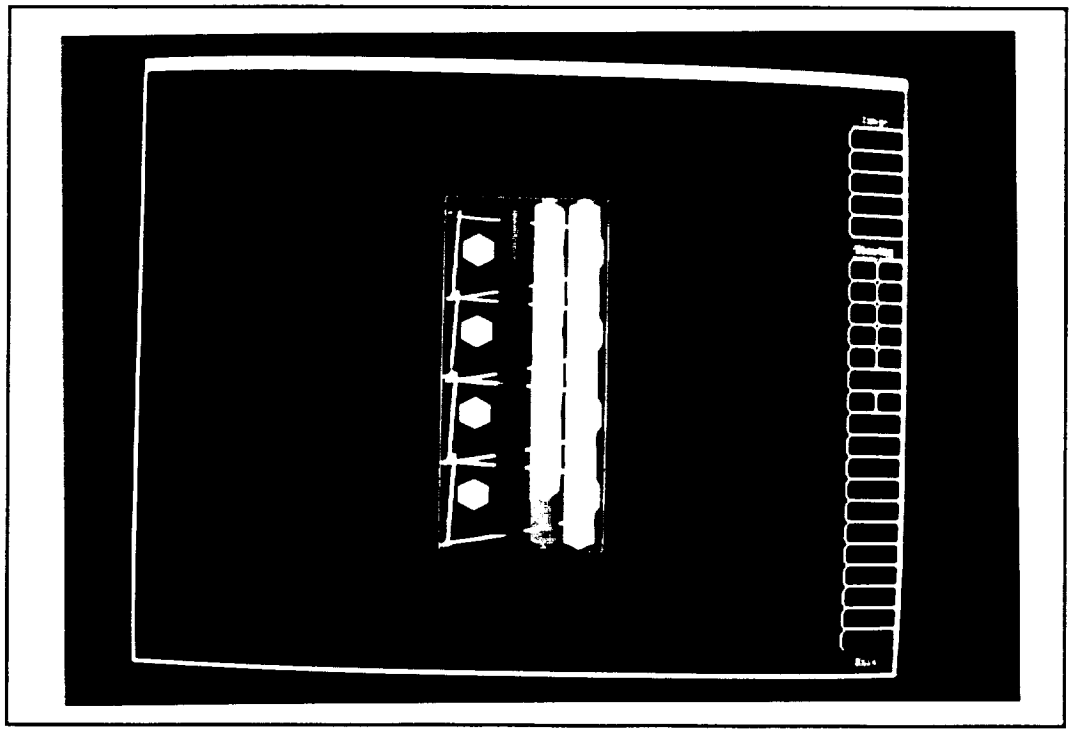


(b) side view

Figure 4.9 (1 0 0) cut plane of a MWK composite.

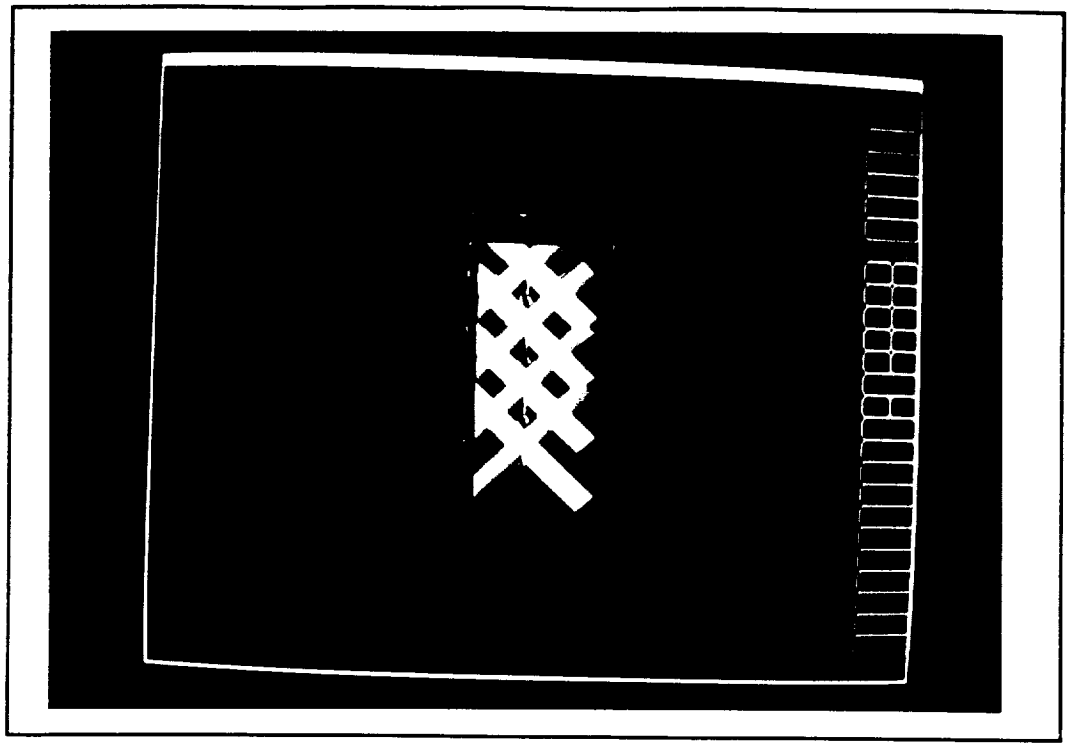


(a) cut section and top view

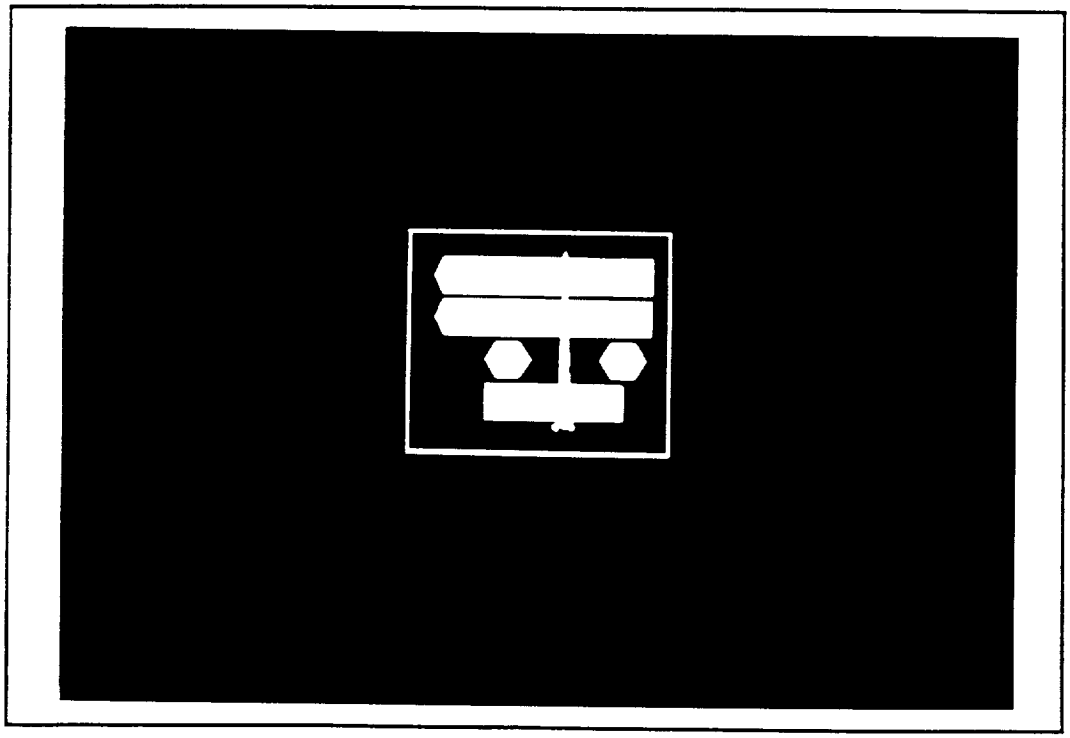


(b) side view

Figure 4.10 (1 0 0) cut plane of a MWK composite. (second cut)

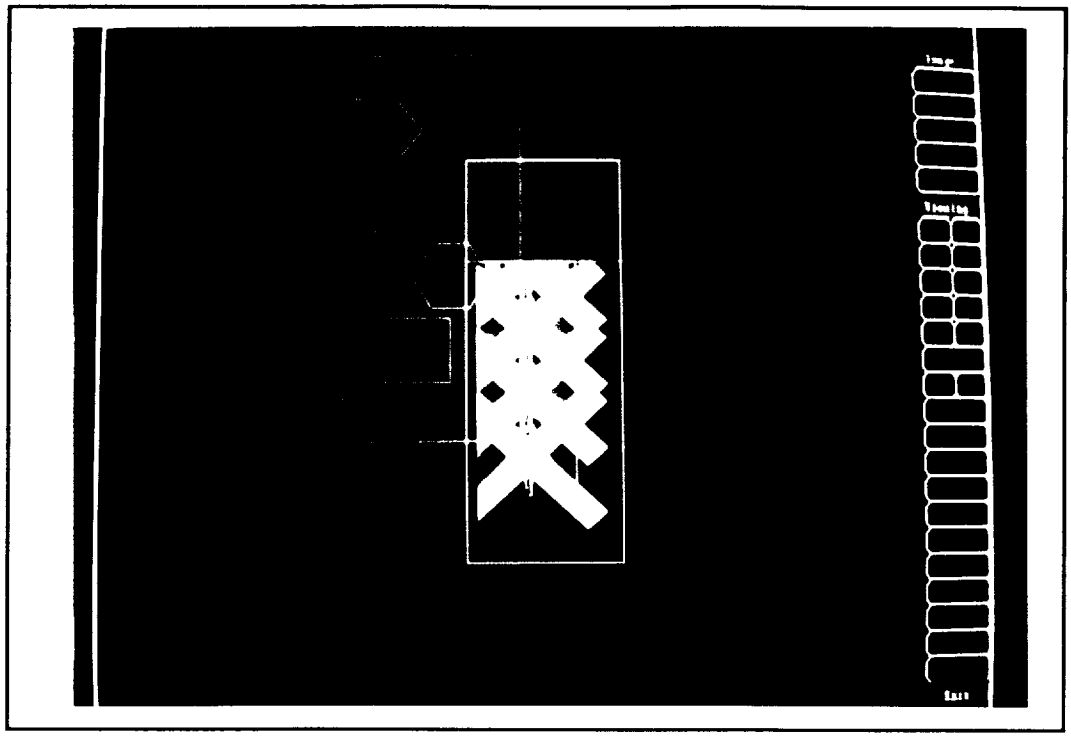


(a) cut section and top view

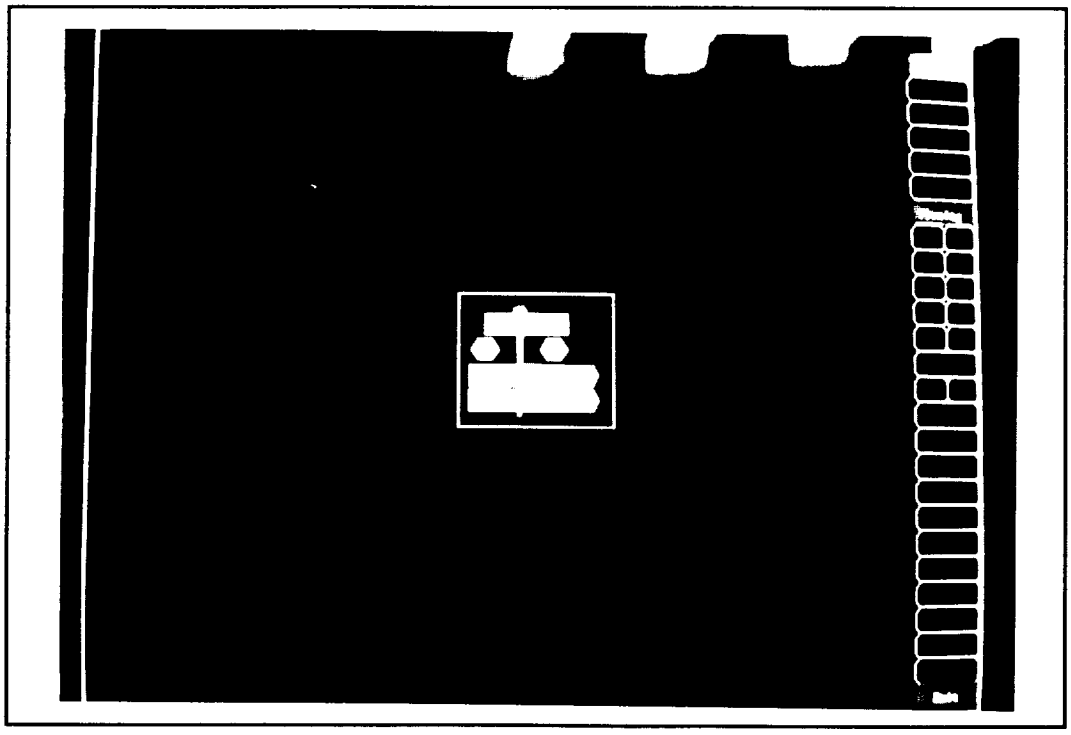


(b) front view

Figure 4.11 (0 1 0) cut plane of a MWK composite.

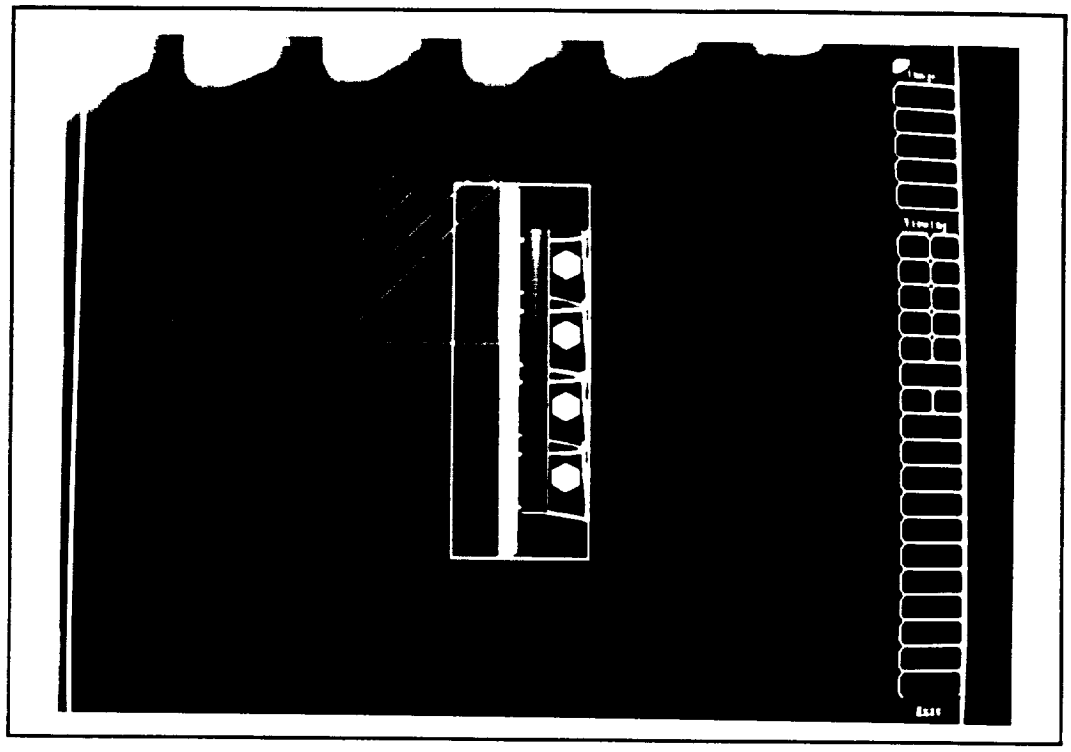


(a) cut section and top view

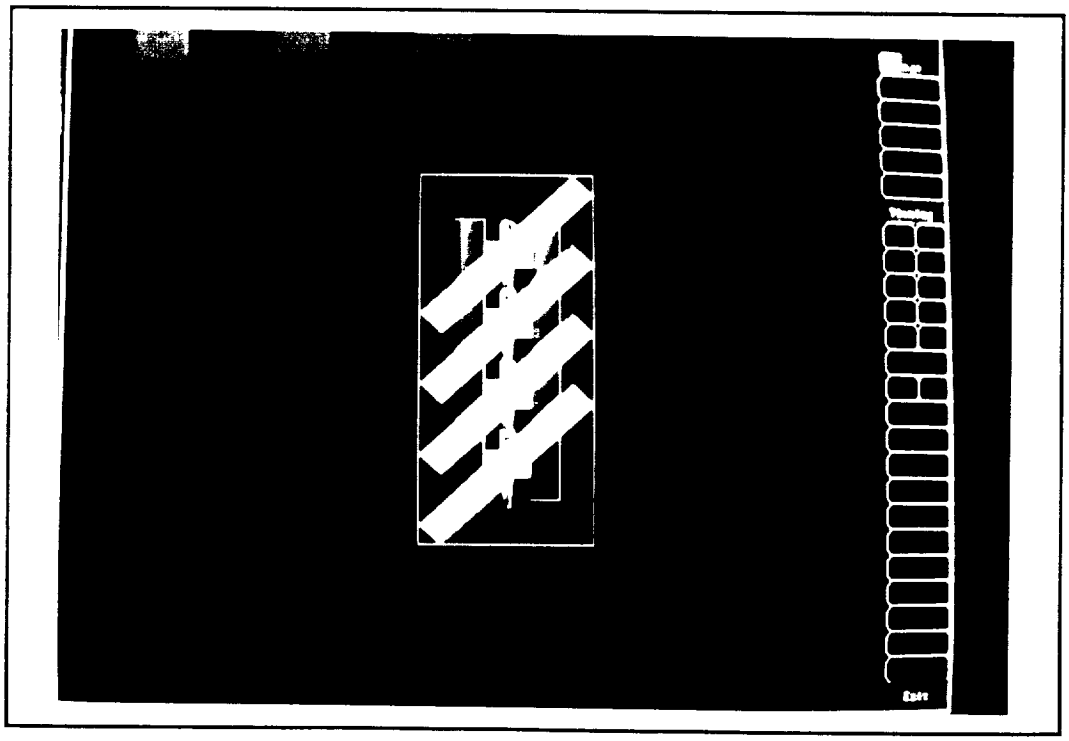


(b) front view

Figure 4.12 (0 1 0) cut plane of a MWK composite. (second cut)

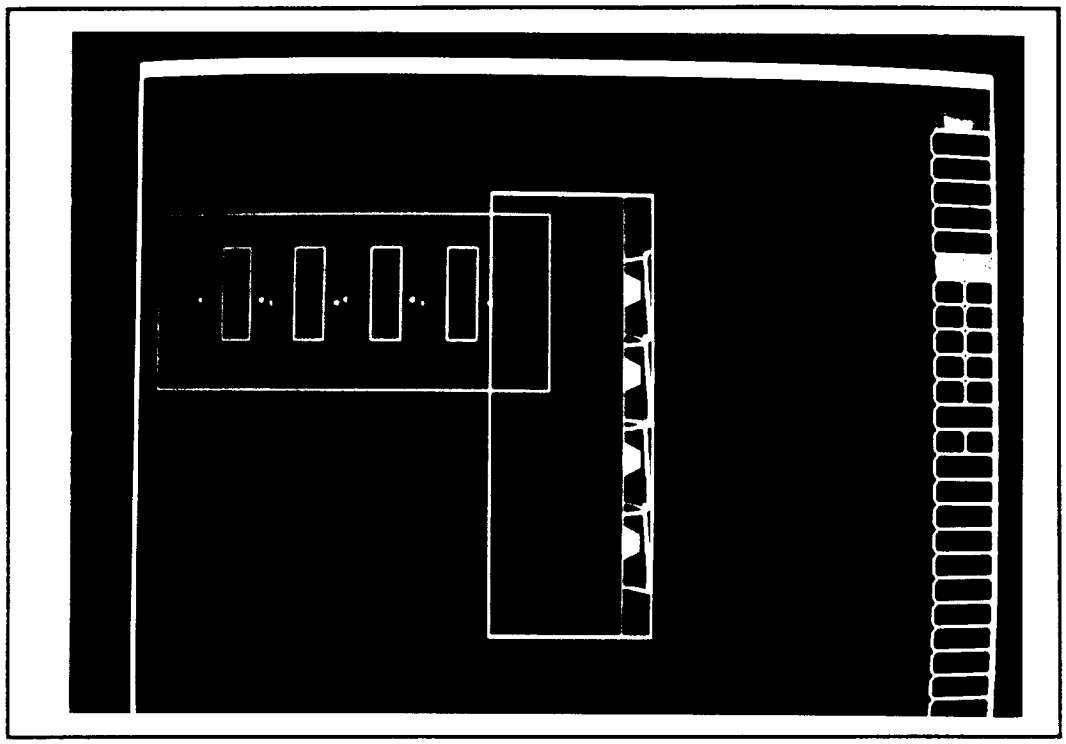


(a) cut section and side view

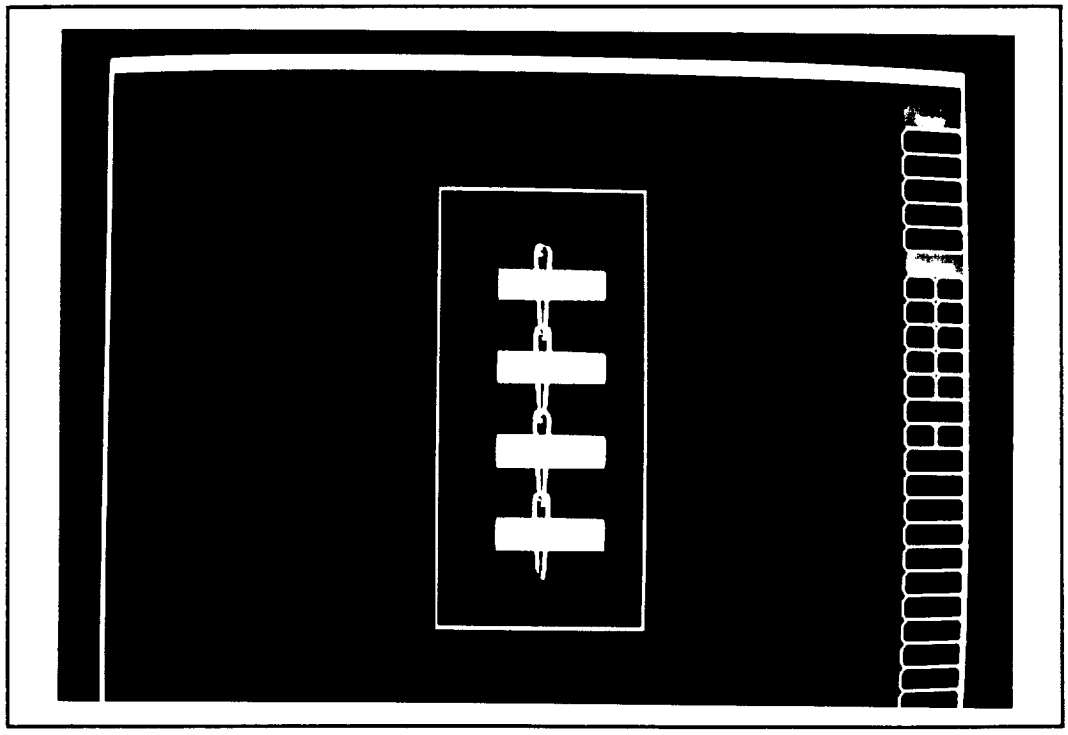


(b) top view

Figure 4.13 (1 0 0) cut plane of a MWK composite.



(a) cut section and side view



(b) top view

Figure 4.14 (1 0 0) cut plane of a MWK composite. (second cut)

4.3.2 Experimental Verification

The schematic of viewing planes for multiaxial warp knit composites are illustrated in Figure 4.15. The montages from the first polishing on each plane are shown in Figure 4.16 - 4.18. In Figure 4.16, 90° - insertion yarns and stitch "knots" with regular spacing are shown as the composite is observed from a top view. Figure 4.17 and Figure 4.18 show the front view and side view of the composite, respectively. The insertion yarns look like a shape of race track due to the compressive pressure in the thickness direction during composite processing. By observations, the unit cell dimension from the montages of the first layer can be defined as 2.2, 2.2 and 6.2 mm, in length, width and thickness, respectively. The montages after the second polishing are shown in Figures 4.19 - 4.21. Comparing Figure 4.16 with Figure 4.19, the 90° layer was polished away; the 45° layer shows up. From the front view, only knots or cross-sections can be seen; while the stitch loop can be observed from side view, as shown in Figure 4.21. Figure 4.22 shows the detailed chain stitch structure (from surface 3). The schematic of the polishing planes is shown in Figure 4.7.

In comparison to the computer generated graphics, Figure 4.9 and Figure 4.10 are similar to Figure 4.16 and Figure 4.19, respectively. Figure 4.11 and Figure 4.12 are similar to Figure 4.17 and Figure 4.20, respectively. Figure 4.13 and Figure 4.14 are similar to Figure 4.18 and Figure 4.21, respectively. As mentioned earlier, the exact shapes of the insertion yarns are controlled by the the processing conditions. However, the relative locations of the insertion yarns can be depicted from the photomicrographs. In conclusion, the unit cell geometry of MWK composites can be described and visualized by the computer software.

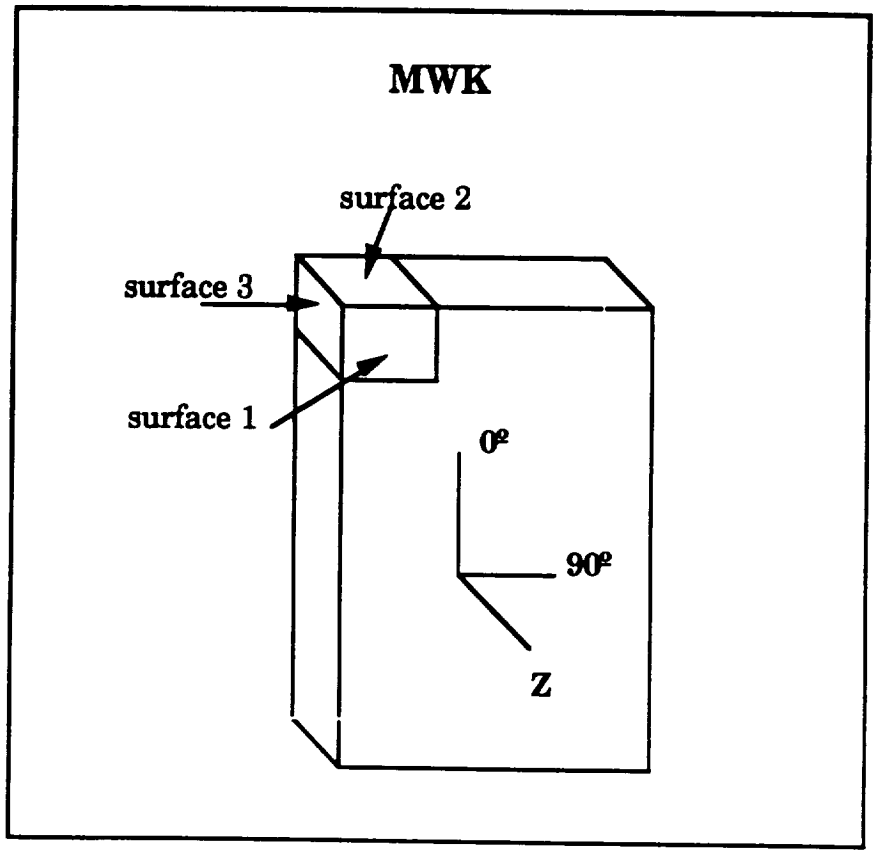


Figure 4.15 Schematic of cutting planes for a MWK composite.

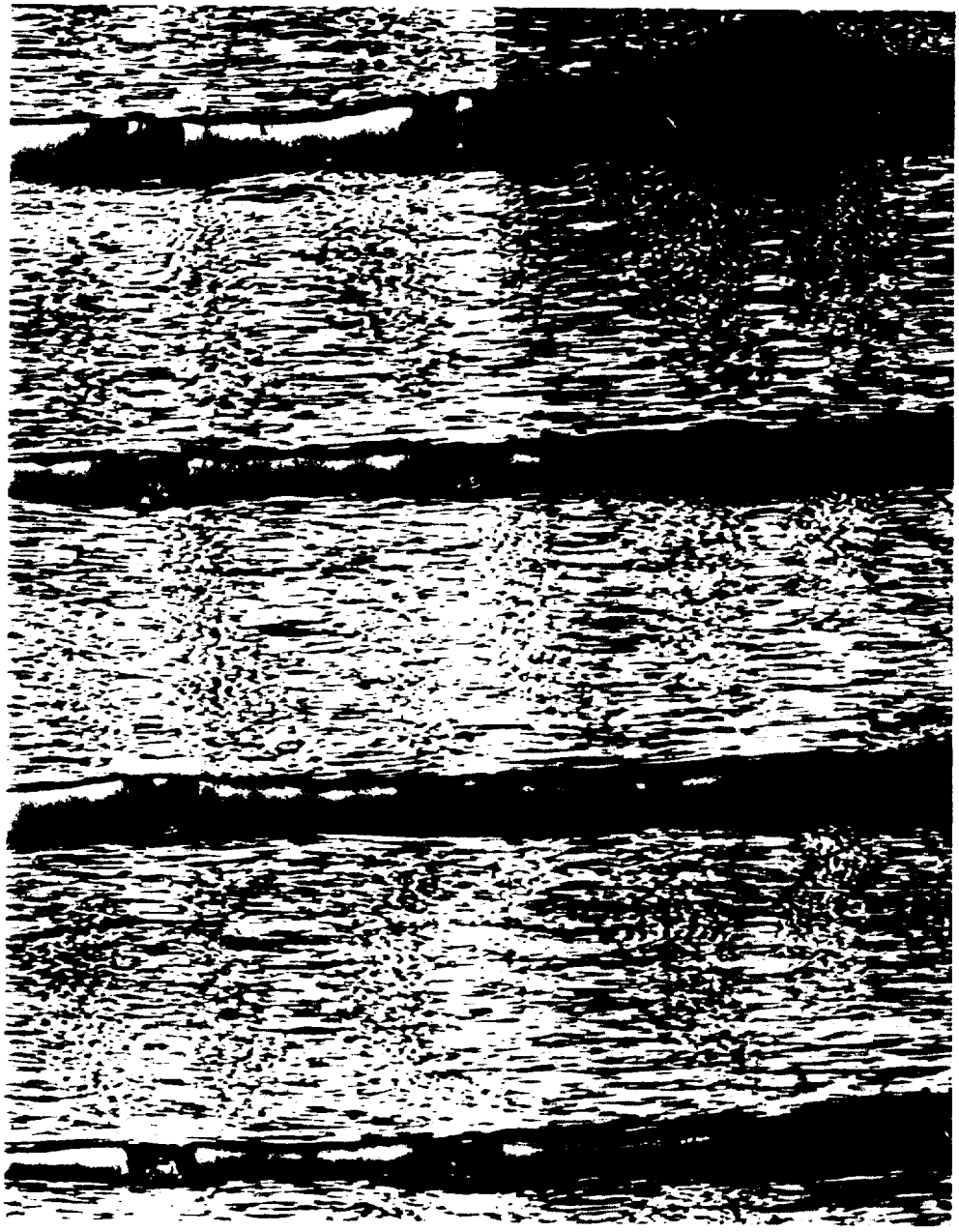


Figure 4.16 Photomicrograph of a MWK composite. (surface 1)

Figure 4.17 Photomicrograph of a MWK composite. (surface 2)

Figure 4.18 Photomicrograph of a MWK composite. (surface 3)

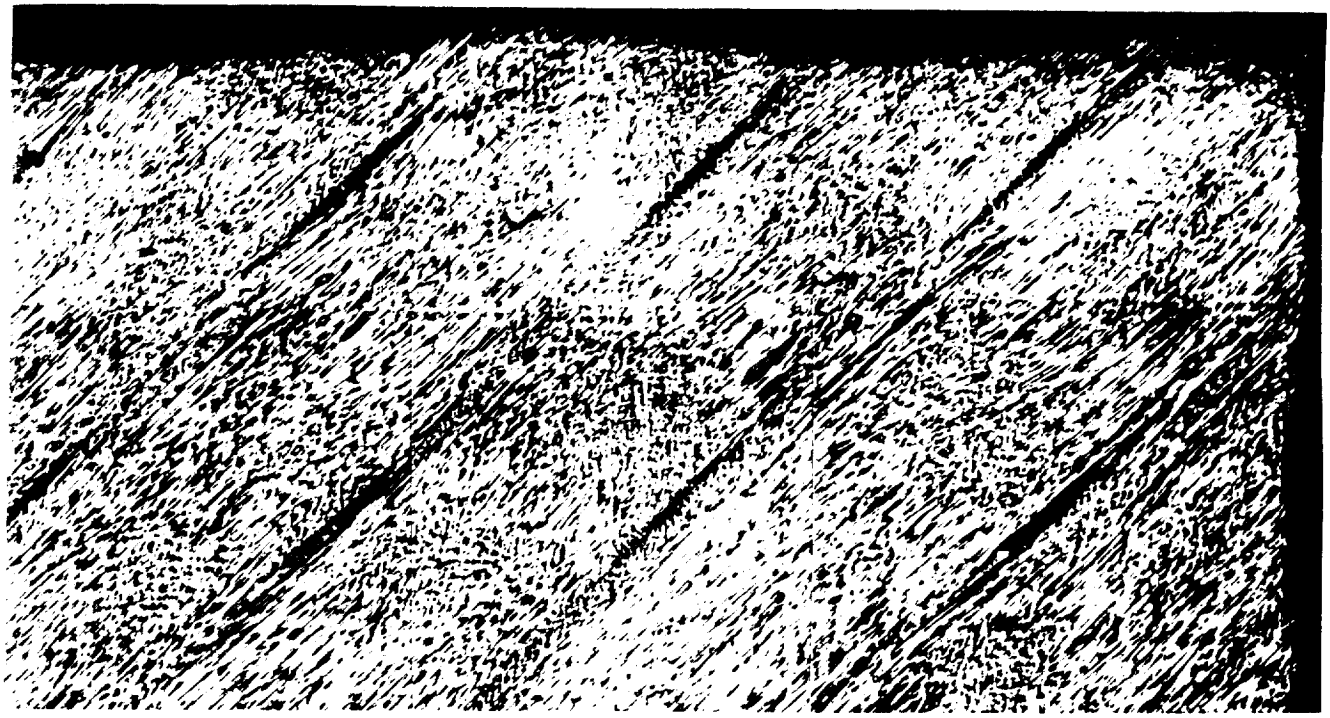


Figure 4.19 Photomicrograph of a MWK composite. (surface 1, second polishing)

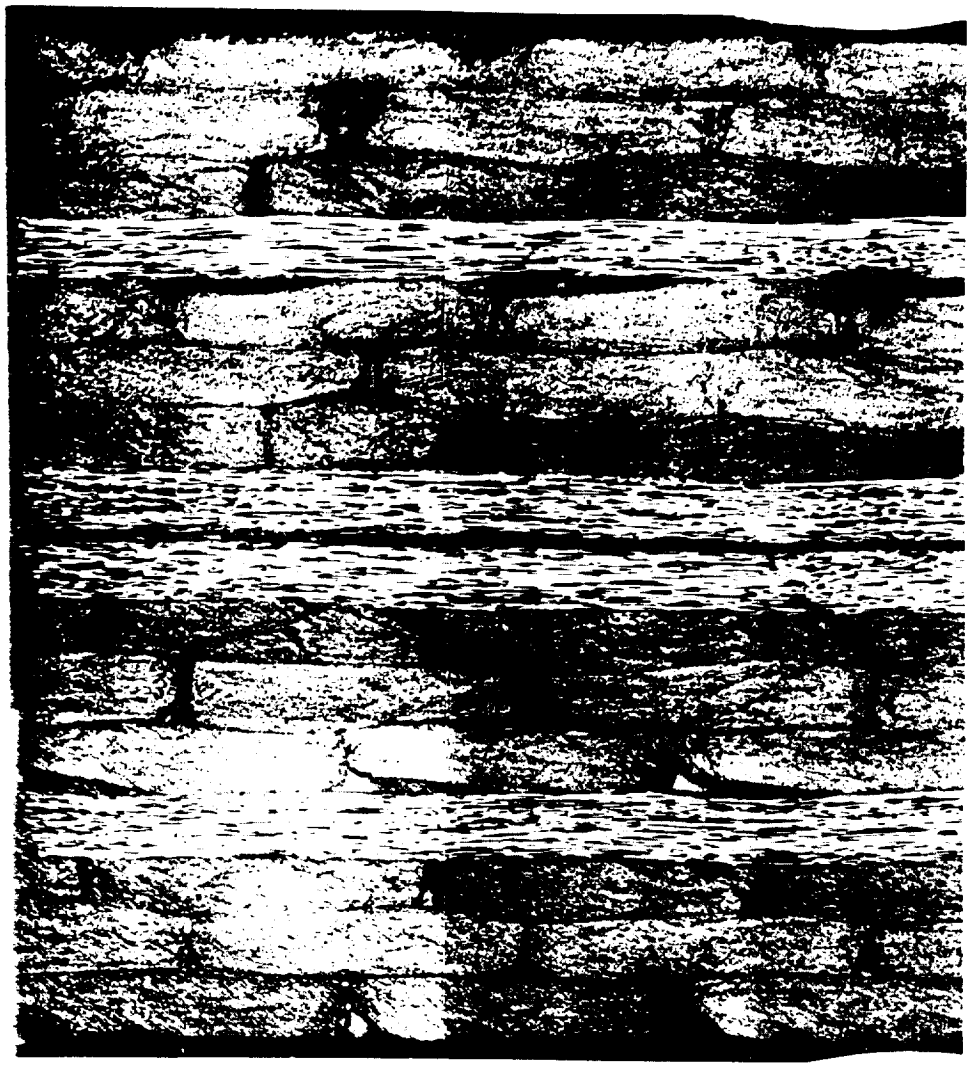


Figure 4.20 Photomicrograph of a MWK composite. (surface 2, second polishing)

Figure 4.21 Photomicrograph of a MWK composite. (surface 3, second polishing)

Figure 4.22 Photomicrograph of a MWK composite, showing the chain stitch loop.

Chapter - 5. Unit cell characterization

5.1 Background

The traditional approach used in modeling of composites reinforced by three-dimensional (3-D) preforms is to assume that their fiber volume fraction and fiber orientation are known, either obtained from experimental measurement or provided by preform manufactures, the relationship between preform fiber architecture and preform processing variables is not considered. In this report, we first examine both 3-D braiding and Multiaxial Warp Knitting in the light of fiber architecture, followed by the development of geometric models for 3-D braided and MWK structures using a unit cell approach. The unit cell geometries of these two 3-D fabrics are identified, and the relationship of structural parameters such as yarn orientation angle and fiber volume fraction with the key processing variables is established. The limiting geometry has been computed by establishing the point at which yarns jam against each other. Using this limiting geometry factor makes it possible to identify the complete range of allowable geometric arrangements for 3-D fabric preforms. The identified unit cell geometries can then be translated to mechanical models which relate the geometrical properties of fabric preforms to the mechanical responses of composite systems.

5.2 3-D Braiding

The unit cell geometry of the track-and-column braid has been investigated by many researchers since the early 1980's [5.1-4]. A common assumption made in most of the analyses is that the braider yarns are oriented along the four diagonals in the unit cell. However, the fiber volume fraction of 3-D braid is normally over 0.5, so the yarns cannot be treated as dimensionless lines to cross each other at the center of the unit cell. This unit cell geometry is either oversimplified or incorrect. Li, Hammad and El-Shiekh [5.3] described a more realistic unit cell geometry, assuming a cylindrical shape for yarns. According to their analysis, at the yarn jamming point, the yarn orientation angle has a maximum value of 55° and the yarn volume fraction has a maximum value of 0.685. Assuming a fiber packing fraction of 0.785, this means that maximum fiber volume fraction for the track and column braid is 0.538. At low braiding angles (≤20°), the geometric model by Li, Hammad and El-Shiekh [5.3] predicts a fiber volume fraction of less than 0.328 (also assuming a fiber packing fraction of 0.785).

In this report, we propose a microgeometric model for the track and column braid based on experimental observations and computer simulation. The unit cell geometry has been defined to establish the relationship of geometric parameters and processing variables.

A summary of braiding process is descried in the following prior to the unit cell modeling. As described in Chapter 2, Figure 5.1(a) shows a basic loom setups in a rectangular configuration. The carriers are arranged in tracks and columns to form the required shape and additional carriers are added to the outside of the array in alternating locations. Four steps of motion are imposed to the tracks and columns during a complete braiding machine cycle, resulting in the alternate X and Y displacement of yarn carriers, as shown in Figure 5.1(b)-(e). Since the track and column both move one carrier displacement in each step, the braiding pattern is referred to as 1x1.

5.2.1 Assumptions and Nomenclature

Following assumptions are made to simplify the geometric model of 3-D track-and-column braided structures:

1. No axial yarns (0° insertion) are included in the 3-D structure. Axial yarns may be used to increase composite modulus and strength in longitudinal direction, although it is not popular in practice due to the fact that 3-D braids usually have low orientation angle as fabricated and hence the strength increase is not phenomenal.
2. 1x1 braiding pattern is assumed. Other patterns such as 1×2 and 2×2 can also be used but are not popular in practice. The techniques used for 1×1 can easily be extended to other patterns.
3. Braider yarns have circular cross-sections, same linear density and constant fiber packing fraction;
4. Yarn tensions during braiding are sufficiently high and hence the yarn crimping effects are negligible.

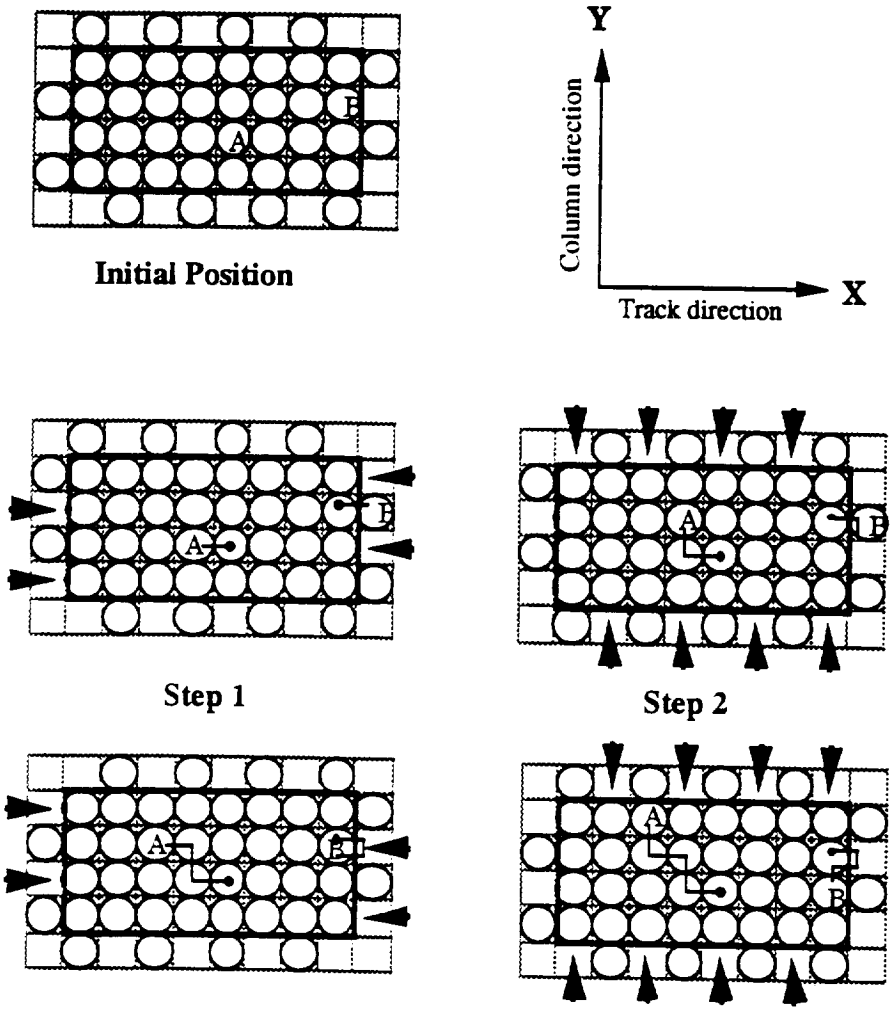


Figure 5.1 - Formation of a rectangular 3-D track and column braid

The definitions of the symbols used in our analysis are listed below:

A_c area of braid cross-section vertical to braid axis

A_f area of total fibers in braid cross-section vertical to braid axis

A_y area of braider yarn in braid cross-section vertical to braid axis

b dimension of yarn cross-section in x'-z and y'-z planes

b' dimension of yarn cross-section in x'-y' plane

d yarn diameter

h_z pitch length of braid formed in a machine cycle (four braiding steps)

N_y number of braider yarns

V_f fiber volume fraction - all fibers to total volume

η braid tightness factor

κ yarn packing fraction (fiber-to-yarn area ratio)

θ angle of braider yarn to braid axis (yarn orientation angle)

Subscripts z, x', y' all refers to co-ordinates.

5.2.2 Unit Cell Geometry

The traditional approach used in modeling 3-D braided composites is to artificially define a unit cell geometry for a 3-D braided structure without providing any relationship between processing variables and geometric parameters [5.4]. All fibers in the unit cell are assumed to incline in 4 different diagonal directions, as well as along the longitudinal direction, if any. Fiber volume fraction is assumed to be either known or measured. In this work the dimension, shape and fiber architecture of the unit cell is based on process and structural analysis. Once the unit cell is identified the relationship between processing variables and key geometric parameters has been established.

The key geometric parameters of 3-D braids (which affect reinforcement capability and composite processability) include braider orientation, total fiber volume fraction, volume

fraction of inter-yarn void and axial fiber percentage of total fibers. Although there are only two simple process parameters adjustable to control the micro-structure of 3-D braids (speed ratio between braiding and take-up and linear density ratio of braider and axial yarns), the process-structure model of 3-D braid is complicated.

Normally, yarn bundles consisting of numerous continuous filaments are used for fabric preforms, thus, the fabric microstructure has three levels: geometry of interfiber packing in the yarn bundle (fiber level), cross-section of yarn bundles in the fabric (yarn level) and orientation and distribution of fibers in the 3-D network (fabric level). The unit-cell technique is commonly used to establish the geometric relation. In most of 2-D fabrics a unit cell geometry is readily identified, but in complex 3-D fabrics it can be very difficult to define.

The fiber volume fraction of a 3-D fabric depends on the level to which yarns pack against each other in the structure and the level to which fibers pack against each other in a yarn. Two basic idealized packing forms can be identified: open-packing, in which the fibers are arranged in concentric layers, as illustrated in Figures 5.2(a) and 5.2(c); and close-packing, in which the fibers are arranged in a hexagonal pattern as in Figures 5.2(b) and 5.2(d).

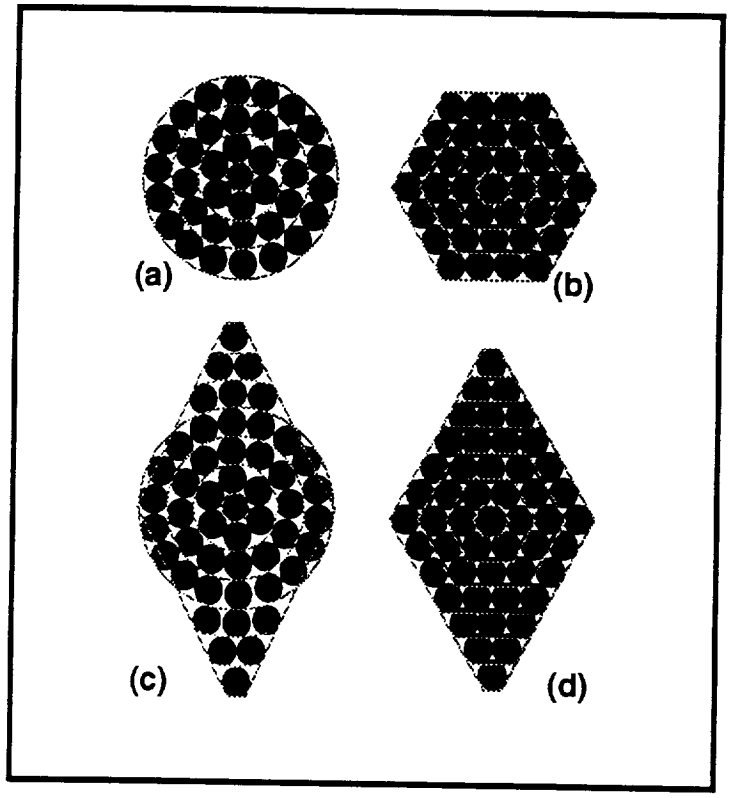


Figure 5.2 Fiber packing in yarns. (a) Open-packing in circular yarns; (b) Close-packing in hexagonal yarns; (c) Open-packing in diamond-shaped yarn; (d) Close-packing in diamond-shaped yarn.

In addition to the level of packing fraction, the fibers also establish the yarn cross-sectional shape, i.e., yarn packing in fabrics. This shape plays a very significant role in determining how many fibers can be packed into a fabric. One of good examples is the yarn packing in 2-step braided preforms [5.5]. Due to the use of untwisted fiber bundles and high braiding tensions, cross-section of axial yarns in the 2-step braid is deformed to prismatic shapes, giving most compact yarn packing within the braided structure. For the track-and-column braids, the braiding tensions are lower compared to the 2-step braids and the cross-sections of yarns actually have a polygonal shape [5.6]. For reasons of simplicity, we idealize the polygonal yarn cross-section as circular shape.

In order to understand the braid internal structure and the yarn interlacing pattern, analysis of braiding carrier motion [5.6], computer graphics simulation [5.7] and computer solid modeling [5.8] was performed and experimental observations of cross-sections of a CARBON-PEEK braided composite[5.6] were made. Figure 5.3 shows an idealized braid cross-section cut longitudinally at a 45° angle to the braid surface. There are four groups of yarns inclined at angle a with the braid axis (z direction) in different directions; the yarns in each group are parallel to each other within a specific plane. Two groups of yarns are parallel to the x'-z plane; the other two are parallel to the y'-z plane. The cutting plane is so selected that it cuts through the diameter of a group of yarns.

Figure 5.4(a) shows the unit cell identified from the analysis. The unit cell consists of four partial yarns being cut by six planes. Clearly, there does not exist such a unit cell which only consists of four complete yarns. The dimensions of the unit cell are $(1/2)h_{x'}$ in x' direction, $(1/2)h_{y'}$ in y' direction and $(1/2)h_z$ in z direction (braid length). The cross-sections of the unit cell at $(1/2)h_z$, $(3/8)h_z$, $(1/4)h_z$, $(1/8)h_z$ and 0, are shown in Figures 5.4(b) - (f), respectively.

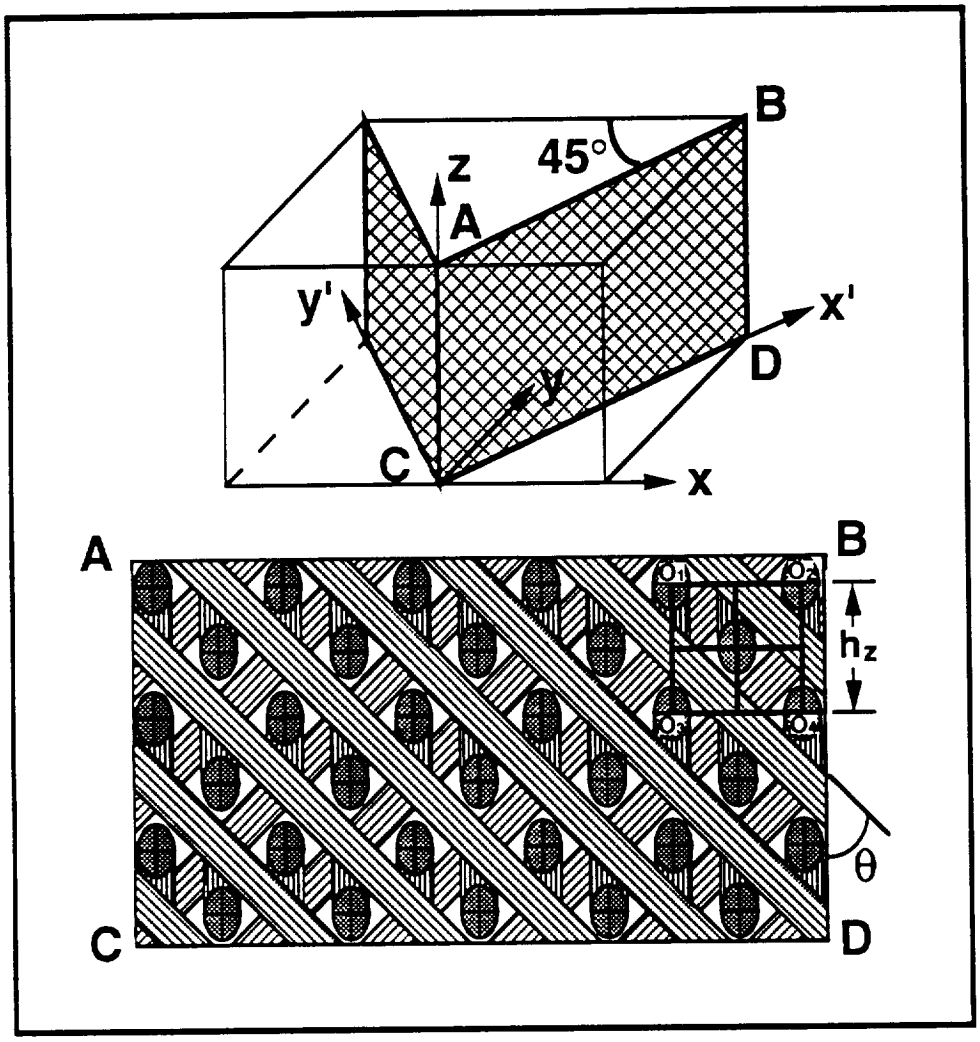


Figure 5.3 Braid cross-section cut longitudinally at a 45° angle to the braid surface by the x'-z plane ABCD. z is the braid length direction.

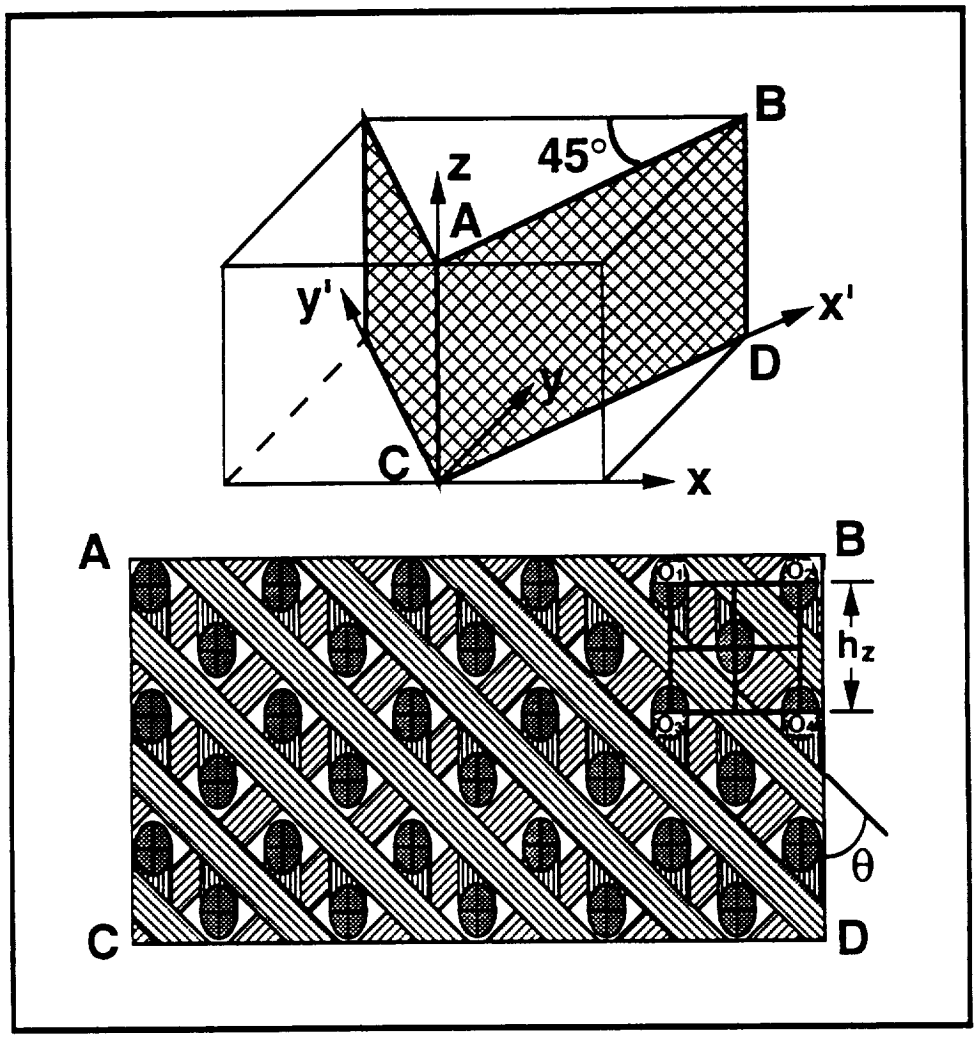


Figure 5.3 Braid cross-section cut longitudinally at a 45° angle to the braid surface by the x'-z plane ABCD. z is the braid length direction.

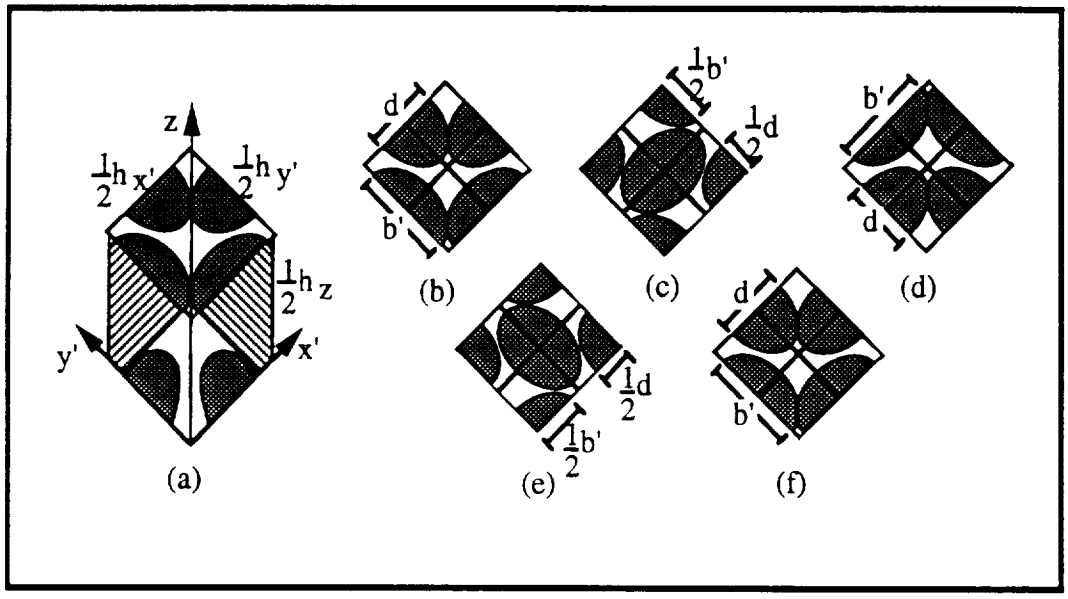


Figure 5.4 Unit cell geometry of 3-D track and column braid. (a) unit cell. (b) unit cell cross-section at z = 1/2 hz. (c) unit cell cross-section at z = 3/8 hz.(d) unit cell cross-section at z = 1/4 hz. (e) unit cell cross-section at z = 1/8 hz. (f) unit cell cross-section at z = 0.

5.2.3 Unit Cell Model

The portion of $O_1O_2O_3O_4$ in Figure 5.3 is extracted and enlarged in Figure 5.5 to show geometric details. In Figure 5.5, d is the yarn diameter, θ is the braiding angle. Dimensions of b, b' (see Figure 5.4) and h_z can be obtained from the trigonometric relationships:

$$b = \frac{d}{\sin\theta} \tag{5-1}$$

$$b' = \frac{d}{\cos\theta} \tag{5-2}$$

$$h_z = 2\,d\,\frac{\sqrt{1+\cos^2\theta}}{\sin\theta} \tag{5-3}$$

h_z is actually the pitch length of braid formed in a complete machine cycle (four steps). This length is one of the key parameters in controlling the fabric microstructures. Using Equation (3), the braiding angle θ can be expressed in terms of the braid pitch length h_z:

$$\theta = \sin^{-1}\sqrt{\frac{8}{(h_z/d)^2+4}} \quad (h_z \geq 2d)$$

Restriction $h_z \geq 2d$ was applied to the above equation to ensure that $0 \leq \theta < 90°$. The physical meaning of this restriction is that the braid pitch length must be greater than two diameters of the yarn to maintain a stable convergence point during braiding process.

As can be seen in Figure 5.4(b)-(f), each unit cell cross-section consists of four half oval cross-sections of yarn. Based on the assumption of non-crimp yarn paths, total yarn area in every braid cross-section must be the same. Therefore, a conclusion can be reached that any unit cell cross-section contains four half oval cross-sections of yarn. The fiber volume fraction can then be easily derived as:

$$V_f = \frac{2\left(\frac{1}{2}d\right)\left(\frac{1}{2}b'\right)\pi}{\left(\frac{1}{2}hx'\right)\left(\frac{1}{2}hy'\right)}\kappa = 2\,\pi\,k\,\frac{d^2}{hx'\,hy'\cos\theta} \tag{5-4}$$

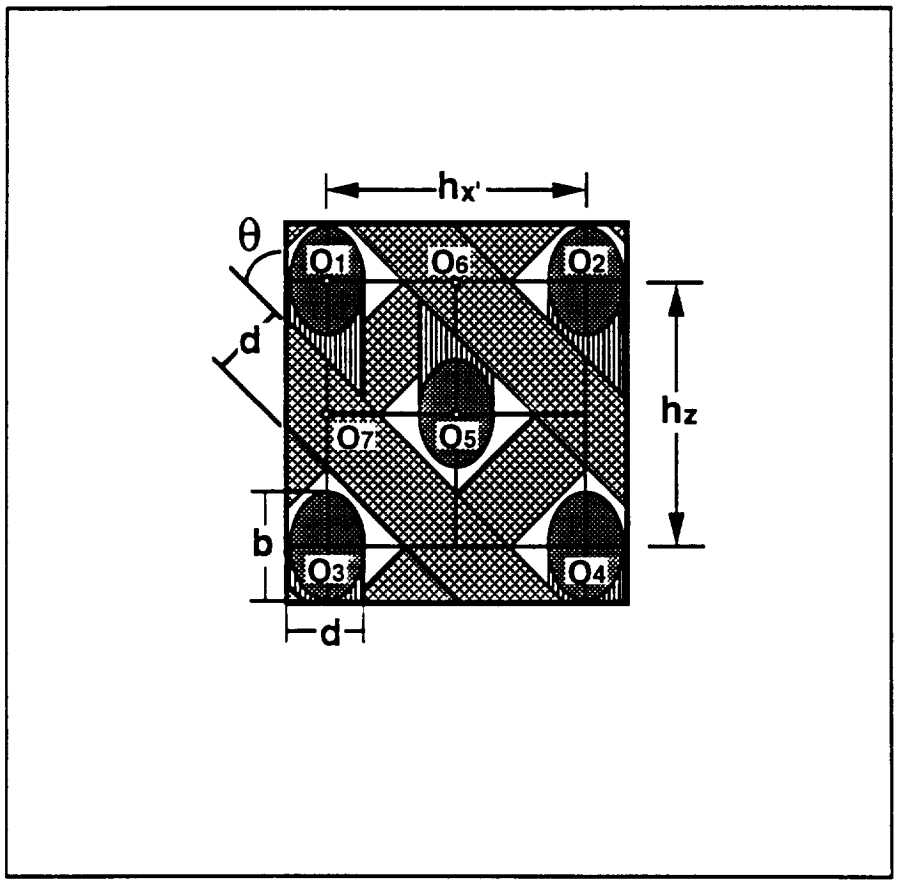


Figure 5.5 A portion of braid cross-section (O1O2O3O4) in the x'-z plane consisting of four unit cells.

where κ is the fiber packing fraction, which normally has a value of about 0.785 for 3-D braids after matrix addition and consolidation [5.5].

The braid has the tightest structure when each yarn is in contact with all its neighboring yarns, in other words, the yarns are jammed against each other. Under this jamming condition, the unit cell has a minimum dimension in both x' and y' directions, i.e.:

$$h_{y'min} = h_{x'min} = 2\,d\,\frac{\sqrt{1+\cos^2\theta}}{\cos\theta} \tag{5-5}$$

and the braid has a maximum fiber volume fraction, which can be derived by combining Equations (4) and (5):

$$V_{fmax} = \frac{\pi}{2}\,\kappa\,\frac{\cos\theta}{1+\cos^2\theta} \tag{5-6}$$

Due to bulky fibers and nonlinear crimp nature, it is difficult to fabricate a braid having the tightest structure. In practice, the fiber volume fraction is calculated directly from the area of fibers to the area of braided composite in the cross-section perpendicular to braid axis:

$$V_f = \frac{A_f}{A_c} \tag{5-7}$$

where A_c is the area of the braided composite and A_f is the area of total fibers given by:

$$A_f = \kappa\,\frac{N_y A_y}{\cos\theta} \tag{5-8}$$

Combining Equations (5-7) and (5-8), we have:

$$V_f = \kappa\,\frac{N_y A_y}{A_c \cos\theta} \tag{5-9}$$

In order to measure the fiber compactness in the braided structure, we define a braid tightness factor, η, as:

$$\eta = \frac{N_y A_y}{A_c} \tag{5-10}$$

According to its definition, the braid tightness factor is actually the ratio of total yarn area to the braid cross-section area in x'-y' plane when $\theta = 0°$. Using Equation (5-10), the fiber volume fraction can be simplified to:

$$V_f = \kappa \frac{\eta}{\cos\theta} \tag{5-11}$$

Clearly, the amount of fibers within the braid is limited by the maximum fiber volume fraction V_{fmax}. Combining Equations (5-6), (5-9) and (5-11), we obtain:

$$V_f = \kappa \frac{\eta}{\cos\theta} \leq \frac{\pi}{2} \kappa \frac{\cos\theta}{1+\cos^2\theta} \tag{5-12}$$

$$\eta \leq \frac{\pi}{2} \frac{\cos^2\theta}{1+\cos^2\theta} \tag{5-13}$$

Based on the geometric analysis given above, braiding angle θ is determined from yarn diameter d and braid pitch length h_z. The fiber volume fraction, V_f, is controlled by braiding angle θ and braid tightness factor η. The tightness factor h, in turn, has to be properly selected so that the required fiber volume fraction V_f is achieved and over-jamming is avoided. Figure 5.6 shows the $V_{f\text{-}\theta}$ relationship based on Equation (5-12).

The fiber packing fraction is assumed to be 0.785 in our calculation. A maximum fiber volume fraction of 0.617 is achieved at $\theta = 0°$ (i.e. the "braid" is an assembly of longitudinal yarns without interlacing). At $\theta = 90°$, which is impossible to achieve either theoretically or in practice, fiber volume fraction approaches zero. There is no such maximum braiding angle as suggested by Li, Hammad and El-Shiekh [5.2], wherein 3-D track-and-column braids can be made with $0 \leq \theta < 90°$. As θ becomes higher, the yarns slide away from each other to form a more open structure, resulting in a lower fiber volume fraction. In practice, a braiding angle higher than 45° is difficult to achieve because of yarn slippage creating difficulties in maintaining a constant height of convergence point.

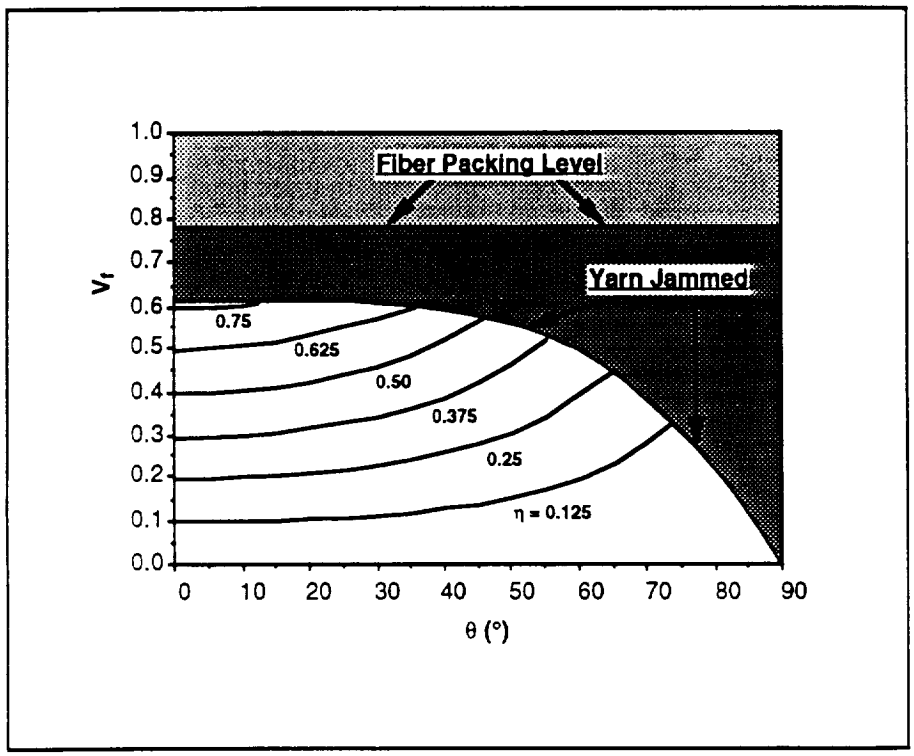


Figure 5.6 Relationship of fiber volume fraction to braiding angle for various tightness factors. Fiber packing fraction κ is assumed to be 0.785.

Although a wide range of fiber volume fractions can be achieved for 3-D track-and-column braids by using different levels of the braid tightness factor, the actual fiber volume fraction is more likely to be closer to the jamming region as shown in Figure 5.6. The reason for this is that yarns are usually tightly compacted by their neighboring yarns due to the high yarn tensions applied to overcome inter-yarn friction during braid formation. In practice, it is possible that the fiber volume fraction achieved will be higher than V_{fmax} under jamming conditions because of non-linear yarn paths (yarn crimp) in the braided structure. For example, at $\theta = 20°$, fiber volume fraction under jamming conditions is 0.615, while fiber volume fractions in composites having a braiding angle of 20° observed from experiments normally fall within a range of 0.60 to 0.65.

5.3 Multiaxial Warp Knit

A series of studies on the technology, structure, and properties of the MWK preforms and composites have been reported by Ko and his co-workers [5.9-13]. In this section, we propose a unit cell model for the four-layer MWK structure. With minor modifications, the analysis can be generalized into the MWK system with six or more layers of insertion yarns.

The MWK fabric preforms having four directional reinforcements similar to quasi-isotropic lay-up can be produced in a single step. The key geometric parameters of the MWK fabric preforms, which affect the reinforcement capability and the composite processability, include the number of yarn axis, the orientation of bias yarns, total fiber volume fraction, pore size and pore distribution, and percentage of stitch fibers to total fiber volume. The processing variables adjustable to control the MWK micro-structure include the type of knit stitch, the ratio of stitch-to-insertion yarn linear density, the orientation angle of bias yarns, and the thread count. The concept of a unit-cell is used to establish the relationship between the geometric parameters and process variables.

5.3.1 Unit Cell Modeling

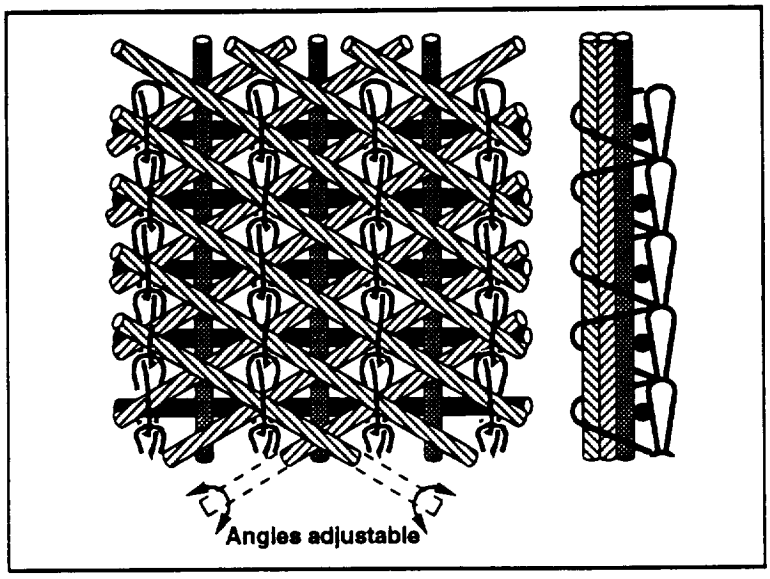
As demonstrated in the development of 3-D braid model, the first step in the unit cell based modeling is to determine the unit cell dimension, so that it is the smallest repeating unit of the structure. The second step is to assume some idealized cross-section shapes of the yarn bundles, based on experimental observations. The final and most important step is to identify the overall unit cell geometry, from which expressions for the key geometric parameters can be derived, and the geometric limits applied to the structure can be defined.

As can be seen in Figure 5.7, the unit cell for the MWK fabric preform can be defined in many ways to meet the definition for a unit cell. However, it would be most reasonable to have a unit cell which consists of a complete knitting stitch, and the insertion yarns in the unit cell are all symmetrical. Figures 5.8(a) and (b) show the unit cells for the MWK structure with chain and tricot stitch, respectively. Within the outlined in the figure, the unit cell consists of one each of 0° and 90° yarns, two each of $+\theta$ and $-\theta$ yarns as well as a knit stitch.

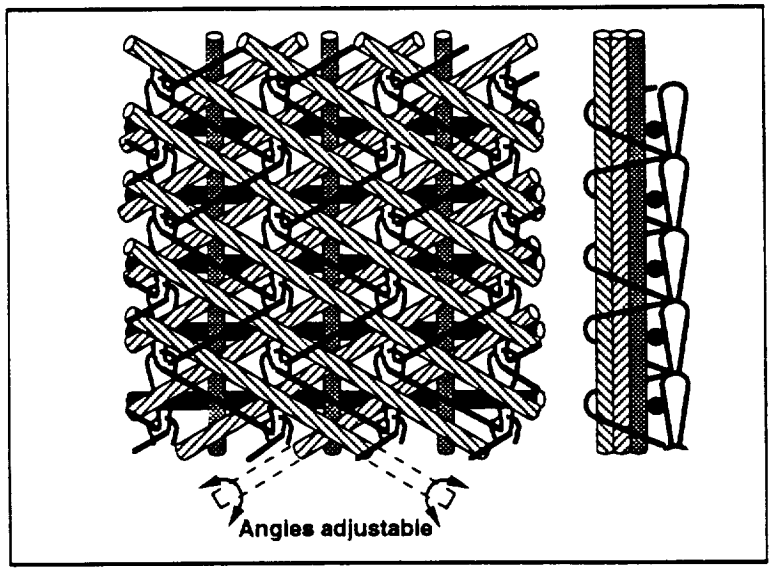
In order to determine the shape of yarn cross-sections, a MWK reinforced carbon/epoxy composite was cut perpendicularly to its 0° axis, polished and examined using an optical microscope. Figure 5.9 shows the photomicrograph of a cross-section of the MWK

composite. As can be seen in Figure 5.9, the insertion yarns have a rectangular cross-sectional shape with rounded corners. For example, the 0° yarns have a width-to-thickness aspect ratio of 5.7±0.42.

Based on this observation, the insertion yarns are assumed to have a race-track cross section having a width-to-thickness aspect ratio $f \geq 1$, as illustrated in Figure 5.10(a). The stitch yarn, which has a much lower linear density than the insertion yarns and usually contains some twist, has less tendency to spread out in the knitting process. Therefore, a circular cross-section with a width-to-thickness aspect ratio $f = 1$ is assumed for the stitch yarn, as illustrated in Figure 5.10(b).

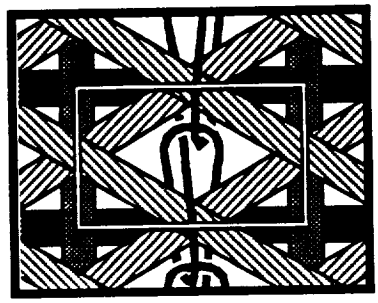


(a) Chain stitch

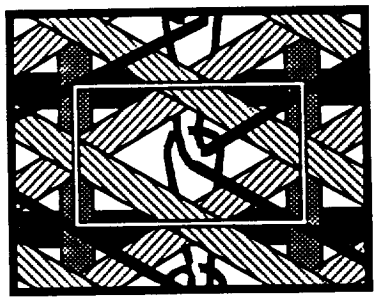


(b) Tricot stitch

Figure 5.7 Multiaxial warp knit (MWK) with four layers (0°, 90°, and ±θ) of insertion yarns and (a) chain stitch or (b) tricot stitch.



(a) Chain stitch



(b) Tricot stitch

Figure 5.8 Unit cell which consists of one each of 0° and 90° yarns, two each of +θ and -θ yarns as well as a knit stitch: (a) with chain stitch; (b) with tricot stitch.

C-2.

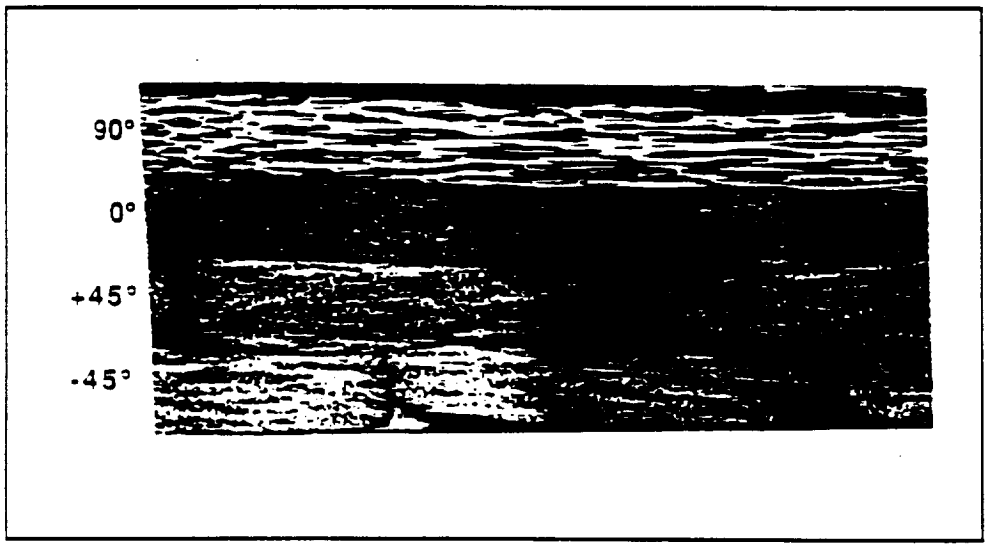


Figure 5.9 Photograph of a cross-section of MWK reinforced carbon/epoxy composite.

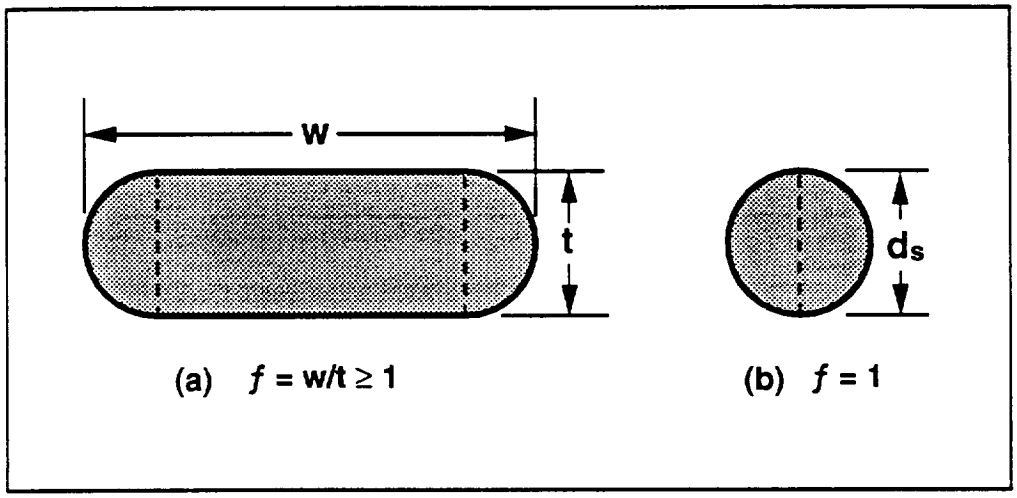


Figure 5.10 Shape and dimension of yarn cross-section: (a) insertion yarn is assumed to have a race-track cross-section with the width-to-thickness aspect ratio $f \geq 1$; (b) stitch yarn is assumed circular in cross-section with $f = 1$.

Figure 5.11 shows the idealized unit cell geometry for the MWK structure, including the shape, dimension, orientation, and position of all the insertion and stitch yarns within the fiber 3-D network. The knit stitch is assumed to have the tightest loop construction, and the curved loop is idealized to a rectangular shape, as illustrated in Figure 5.11(b) and Figure 5.11(d). The dimensions of the unit cell are X, Y, and Z, respectively, corresponding to the 0° axis, 90° axis, and the thickness axis vertical to the 0°-90° plane, as shown in Figure 5.11(a) and Figure 5.11(c).

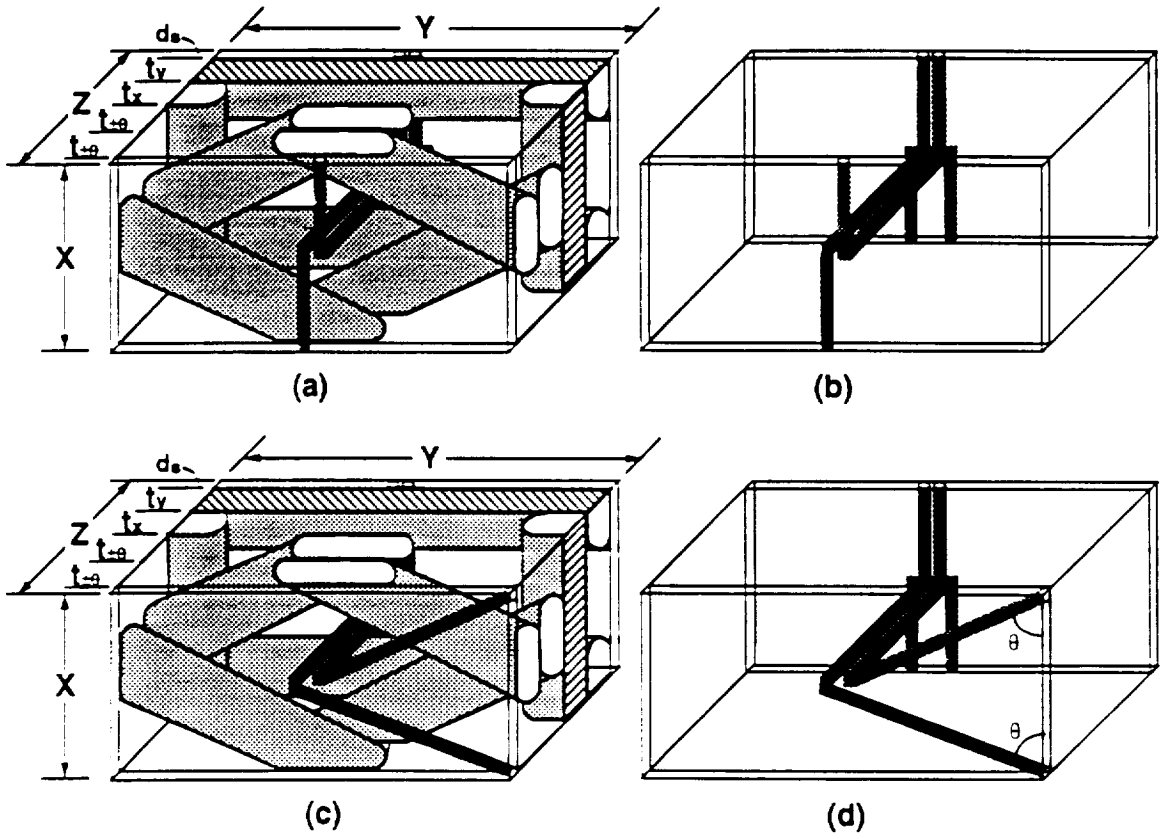


Figure 5.11 Unit cell geometry of MWK structure: (a) with insertion and chain stitch yarns; (b) chain stitch yarns only; (c) with insertion and tricot stitch yarns; (d) tricot stitch yarns only.

Based on the idealized yarn shape and unit cell geometry, the geometric model of the MWK fabric preform can be established. In the following section, the expressions for yarn dimensions, unit cell dimensions, yarn volumes, yarn lengths, fiber volume fractions, and criteria for geometric limit will be presented without derivation.

5.3.2 Nomenclature

κ	fiber packing fraction (fiber-to-yarn area ratio)
ρ	fiber density (kg m^{-3})
λ	yarn linear density (kg m^{-1})
θ	angle of bias yarns to 0° yarn (x direction)
ds	diameter of stitch yarn
w, t	width and thickness of insertion yarns
f	aspect ratio of insertion yarns (width to thickness)
S	area of yarn cross-section (m^2)
L	yarn length (m)
l	yarn length normalized with 0° yarn length (X)
X, Y, Z	unit cell dimensions in height, width, and thickness directions, respectively
ϕ	thread count (number of insertion yarn per unit length, end m^{-1})
η	fabric tightness factor (as defined by Eq.(5-44))
V	volume (m^3)
Vf	fiber volume fraction
v	percentage of insertion or stitch fibers to total fiber volume (%)

Subscripts x, y, z, and $\pm\theta$ all refer to yarn orientations in the unit cell frame

Subscripts c, i, and s refer to unit cell, insertion yarn and stitch yarn, respectively

5.3.3 Dimensions of insertion and stitch yarns.

The widths of insertion yarns:

$$w_x = f_x \cdot t_x \tag{5-14}$$

$$w_y = f_y \cdot t_y \tag{5-15}$$

$$w_{\pm\theta} = f_{\pm\theta} \cdot t_{\pm\theta} \tag{5-16}$$

The thicknesses of insertion yarns:

$$t_x = \sqrt{\frac{\lambda_x}{\kappa_i \left(\frac{\pi}{4} + f_x\right) \rho_x}} \tag{5-17}$$

$$t_y = \sqrt{\frac{\lambda_y}{\kappa_i \left(\frac{\pi}{4} + f_y\right) \rho_y}} \tag{5-18}$$

$$t_{\pm\theta} = \sqrt{\frac{\lambda_{\pm\theta}}{\kappa_i \left(\frac{\pi}{4} + f_{\pm\theta}\right) \rho_{\pm\theta}}} \tag{5-19}$$

The diameter of stitch yarn:

$$d_s = \sqrt{\frac{\lambda_s}{\frac{\pi}{4} \kappa_s \rho_s}} \tag{5-20}$$

The cross-sectional areas of insertion and stitch yarns:

$$S_x = \left(\frac{\pi}{4} + f_x - 1\right) \cdot t_x^2 \tag{5-21}$$

$$S_y = \left(\frac{\pi}{4} + f_y - 1\right) \cdot t_y^2 \tag{5-22}$$

$$S_{\pm\theta} = \left(\frac{\pi}{4} + f_{\pm\theta} - 1\right) \cdot t_{\pm\theta}^2 \tag{5-23}$$

$$S_s = \frac{\pi}{4} \cdot d_s^2 \tag{5-24}$$

5.3.4 Dimensions of unit cell.

The dimensions of the unit cell in x (0°), y (90°) and z (thickness) directions:

$$X = \phi_x \cdot w_y \tag{5-25}$$

$$Y = \phi_y \cdot w_x \tag{5-26}$$

$$Z = t_x + t_y + 2\,(t_{\pm\theta} + d_s) \tag{5-27}$$

The orientation angle of bias yarns:

$$\theta = \tan^{-1}\left(\frac{Y}{X}\right) \tag{5-28}$$

The total volume of the unit cell:

$$V_c = X \cdot Y \cdot Z \tag{5-29}$$

5.3.5 Volumes of yarns in the unit cell.

The volumes of the insertion yarns in x, y and ±θ directions:

$$V_{ix} = S_x \cdot X \tag{5-30}$$

$$V_{iy} = S_y \cdot Y \tag{5-31}$$

$$V_{i\pm\theta} = 2 \cdot S_{\pm\theta} \sqrt{X^2 + Y^2} \tag{5-32}$$

The total volume of the insertion yarns in the unit cell:

$$V_i = V_{ix} + V_{iy} + V_{i\pm\theta} \tag{5-33}$$

The volumes of the chain stitch yarns in x, y and ±θ directions:

$$V_{sx} = 3 \cdot S_s \cdot (X - \frac{1}{2} d_s) \tag{5-34-a}$$

$$V_{sy} = 2 \cdot S_s \cdot d_s \tag{5-35-a}$$

$$V_{sz} = 2 \cdot S_s \cdot (Z - d_s) \tag{5-36-a}$$

$$V_{s\pm\theta} = 0 \tag{5-37-a}$$

The volumes of the tricot stitch yarns in x, y and ±θ directions:

$$V_{sx} = 2 \cdot S_s \cdot (X - \frac{1}{2} d_s) \quad (5\text{-}34\text{-}b)$$

$$V_{sy} = 2 \cdot S_s \cdot d_s \quad (5\text{-}35\text{-}b)$$

$$V_{sz} = 2 \cdot S_s \cdot (Z - d_s) \quad (5\text{-}36\text{-}b)$$

$$V_{s\pm\theta} = S_s \left(\sqrt{X^2 + Y^2} - d_s\right) \quad (5\text{-}37\text{-}b)$$

The total volumes of the stitch yarns in the unit cell:

$$V_s = V_{sx} + V_{sy} + V_{sz} + V_{s\pm\theta} \quad (5\text{-}37)$$

5.3.6 Normalized yarn lengths in the unit cell.

The lengths of insertion yarns normalized with the unit cell dimension X:

$$l_{ix} = \frac{X}{X} = 1 \quad (5\text{-}39)$$

$$l_{iy} = \tan\theta \quad (5\text{-}40)$$

$$l_{i\pm\theta} = 4\sqrt{1 + \tan^2\theta} \quad (5\text{-}41)$$

The length of chain stitch yarn normalized with the unit cell dimension X:

$$l_s = 3 + 2\frac{3 \cdot d_s + Z}{X} \quad (5\text{-}42\text{-}a)$$

The length of chain stitch yarn normalized with the unit cell dimension X:

$$l_s = 2 + 2\frac{3 \cdot d_s + Z}{X} + \sqrt{1 + \tan^2\theta} \quad (5\text{-}42\text{-}b)$$

5.3.7 Fiber volume fractions.

The volume fractions of total insertion and stitch fibers:

$$V_{fi} = \frac{V_i}{V_c} \cdot \kappa_i \tag{5-43}$$

$$V_{fs} = \frac{V_s}{V_c} \cdot \kappa_s \tag{5–44}$$

The volume fractions of all fibers in x, y, z, and $\pm\theta$ directions:

$$V_{fx} = \frac{\kappa_i \cdot V_{ix} + \kappa_s \cdot V_{sx}}{V_c} \tag{5-45}$$

$$V_{fy} = \frac{\kappa_i \cdot V_{iy} + \kappa_s \cdot V_{sy}}{V_c} \tag{5-46}$$

$$V_{fz} = \frac{\kappa_s \cdot V_{sz}}{V_c} \tag{5-47}$$

$$V_{f\pm\theta} = \frac{\kappa_i \cdot V_{i\pm\theta} + \kappa_s \cdot V_{s\pm\theta}}{V_c} \tag{5-48}$$

The overall fiber volume fraction of the MWK fabric preforms:

$$V_f = V_{fi} + V_{fs}, \quad \text{or} \tag{5-49-a}$$

$$V_f = V_{fx} + V_{fy} + V_{fz} + V_{f\pm\theta} \tag{5-49-b}$$

Percentages of insertion and stitch fibers to total fiber volume:

$$v_i = \frac{Vf_i}{V_f} \cdot 100\% \tag{5-50}$$

$$v_s = \frac{Vf_s}{V_f} \cdot 100\% \tag{5-51}$$

5.3.8 Geometric Limitation

With the geometric model established, one can now proceed to examine the geometric limits of the MWK structure. One important factor to be considered in the design and manufacturing of fabrics is the yarn jamming condition in fabrics. Yarn jamming condition is a geometric boundary condition under which all the yarns are touching each other, resulting in the tightest possible structure. The boundary condition which defines the limiting geometry of the MWK fabric can be summarized by Eqs (5-52) to (5-54):

$$X \geq w_y + d_s \tag{5-52}$$

$$Y \geq w_x + 2 \cdot d_s \tag{5-53}$$

$$X \cdot \sin\theta + d_s\,(1 + \cos\theta) \geq w_{\pm\theta} \tag{5-54}$$

Combining with Eqs (25) and (26), Eqs (52) to (54) become:

$$\phi_x \geq \max.\left(1 + \frac{d_s}{w_y},\ \frac{w_{\pm\theta} - d_s\,(1 + \cos\theta)}{w_y \cdot \sin\theta}\right) \tag{5-55}$$

$$\phi_y \geq 1 + \frac{2 \cdot d_s}{w_x} \tag{5-56}$$

Eqs (55) and (56) serve as the criterion for yarn jamming, which not only gives the boundary condition for the geometric model, but also defines the important processing limit. In order to measure the degree of the fabric tightness, a processing parameter - the tightness factor η is introduced:

$$\eta = \frac{(1 + \frac{d_s}{w_y})}{\phi_x} \leq 1 \tag{5-57}$$

Clearly, the higher is the tightness factor, the more compact the insertion yarns are within the fabric structure. When $\eta = 1$, the unit cell has a minimum dimension in X direction, and the fabric has the tightest structure at an given angle of bias yarns. After determining the value of the tightness factor η, the orientation angle of the bias yarns (θ), and the ratio of stitch-to-insertion yarn linear density (λ_s/λ_i) must be so selected that both Eqs (5-55) and (5-56) are satisfied.

5.3.9 Results and Discussion

Using the equations developed in the analysis, we have established the key relations between process variables and fabric geometry. The results of the geometric modeling can be represented in many ways, and the selection of relations to be evaluated depends on the specific technical need. We have focused on the overall fiber volume fraction (V_f), the percentage of stitch fibers (v_s), and the geometric limit to describe the MWK structure. The key process variables include the orientation angle of bias yarns (θ), the ratio of stitch-to-insertion yarn linear density (λ_s/λ_i), and the fabric tightness factor (η),

The relationship between these geometric parameters and process variables is shown in Figures 5.12 -5.14, using Eqs (5-14) to (5-57). In our calculations, we assume that tricot stitch is used, fiber material and fiber packing fraction are the same for all insertion and stitch yarns, i.e., $\kappa = \kappa_i = \kappa_s$ and $\rho = \rho_i = \rho_s$; and all the insertion yarns have the same linear density and width-to-thickness aspect ratio, i.e., $\lambda_i = \lambda_{ix} = \lambda_{iy} = \lambda_{i\pm\theta}$ and $f_i = f_{ix} = f_{iy} = f_{i\pm\theta}$.

5.3.10 Effect of yarn linear density ratio on fiber volume fraction.

The fiber volume fraction relation in Figure 5.12 shows that for the fixed parameters selected, only a limited window exists for the MWK fabric construction. The window is bounded by two factors: yarn jamming and the point of 90° bias yarn angle. Fabric constructions corresponding to the curve marked "jamming" are at their tightest allowable point, and constructions at the $\theta = 90°$ curve have the most open structure. When $\theta < 30°$, jamming occurs in the whole range of yarn linear density ratio from zero to infinite. When θ is in the range of 30° to 40°, the fiber volume fraction decrease with the increase in yarn linear density ratio until jamming occurs. When $\theta = 45°$, the fiber volume fraction decreases with the increase in yarn linear density ratio to a minimum at about $l_s/l_i = 1$, and starts to increase until jamming occurs. When $\theta \geq 60°$, the fiber volume fraction has the same trend as when $\theta = 45°$, but yarn jamming never occurs. The fiber packing in the yarns, taken as 0.75, limits the maximum fiber volume fraction in the fabric.

5.3.11 Effect of fiber orientation on fiber volume fraction.

The fiber volume fraction is plotted in Figure 5.13 against the orientation angle of bias yarns for seven levels of the fabric tightness factor. It can be seen that for a given tightness factor, the fiber volume fraction decreases as the bias yarn angle increases. At $\eta > 1$, yarn jamming always occurs. At $\eta = 1$ and $\theta = 50°$, the fabric has the tightest structure (maximum fiber volume fraction) for the parameters indicated in the caption. At lower levels of η, smaller bias yarn angles are possible, but the fiber volume fractions also become smaller.

5.3.12 Effect of yarn linear density ratio on fractional stitch fiber volume.

Figure 5.14 shows the percentage of tricot stitch fibers as a function of the ratio of stitch-to-insertion yarn linear density. The possible range of bias yarn angle is from 30° to less than 90°. As can be seen in Figure 5.15, the stitch fiber percentage increases as the yarn

linear density ratio increases, but decreases as the bias yarn angle increases. The process window of v_s is bounded by two curves as shown in Figure 5.14. The upper bound is given by the yarn jamming limit, and the lower bound is reached at θ =90°. This provides a guide to determine the yarn linear density ratio to achieve a required stitch fiber percentage for a given bias yarn angle. For example, if we want to fabricate a MWK fabric preform which has a 10% of stitch fibers and 45° of bias yarn angle with other parameter being the same as given in the caption of Fig.5.2, the ratio of stitch-to-insertion yarn linear density required is about 0.15 as can be read directly from Figure 5.14.

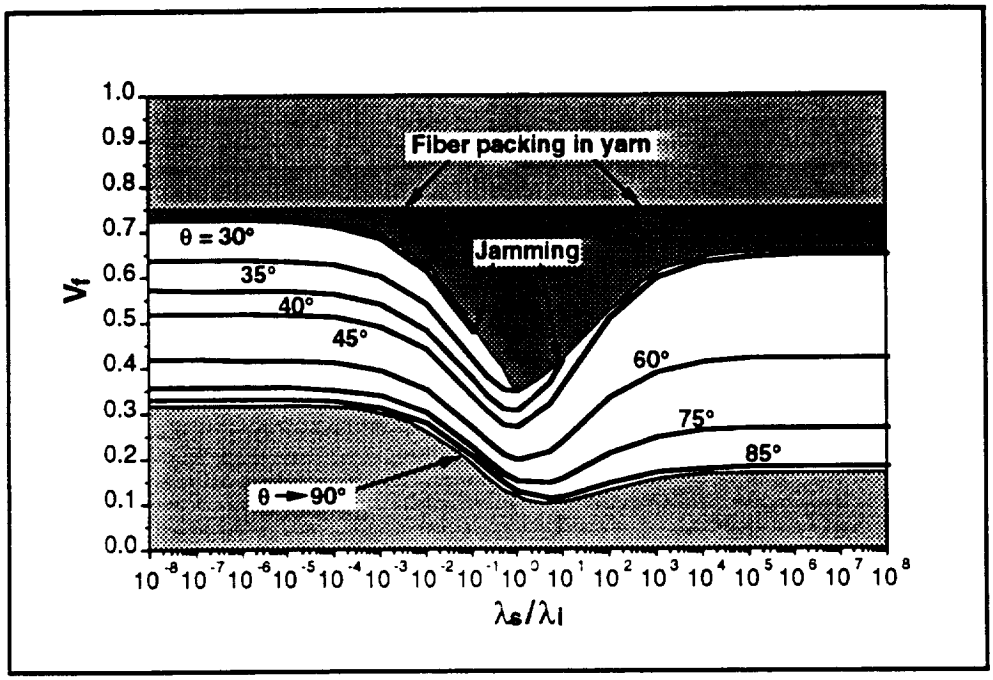


Figure 5.12 Fiber volume fraction versus ratio of stitch-to-insertion yarn linear density (tricot stitch, $\kappa = 0.75$, $\rho = 2.5$ kg m-3, $fi = 5$, and $\eta = 0.5$).

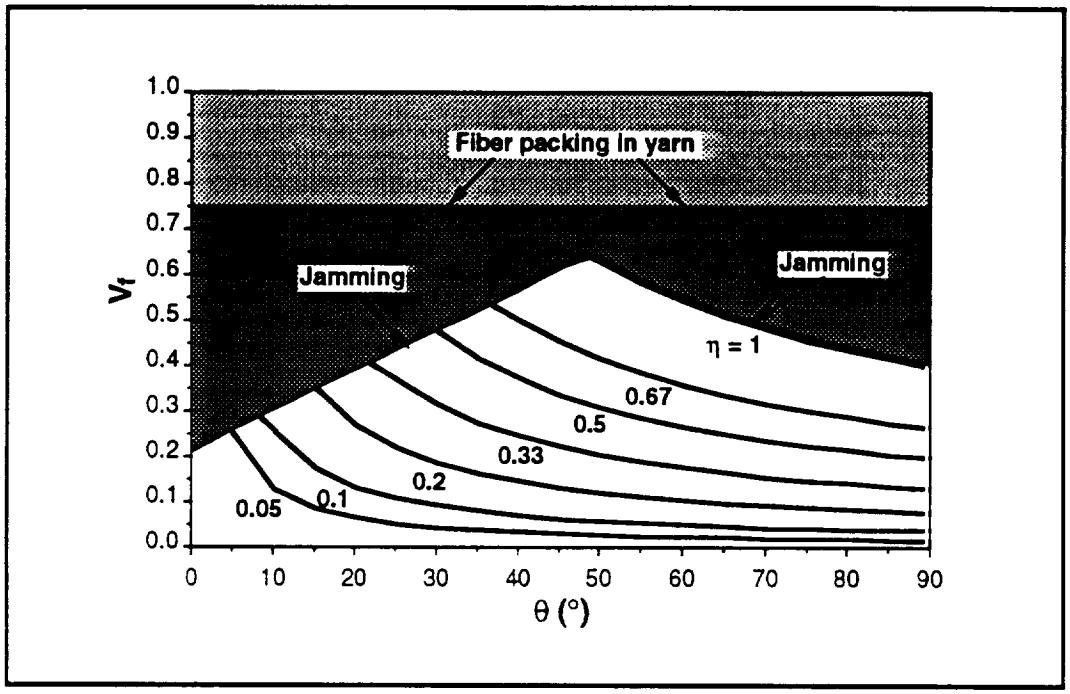


Figure 5.13 Fiber volume fraction versus bias yarn orientation (tricot stitch, $\kappa = 0.75$, $\rho = 2.5$ kg m-3, $fi = 5$, and ls/li = 0.1).

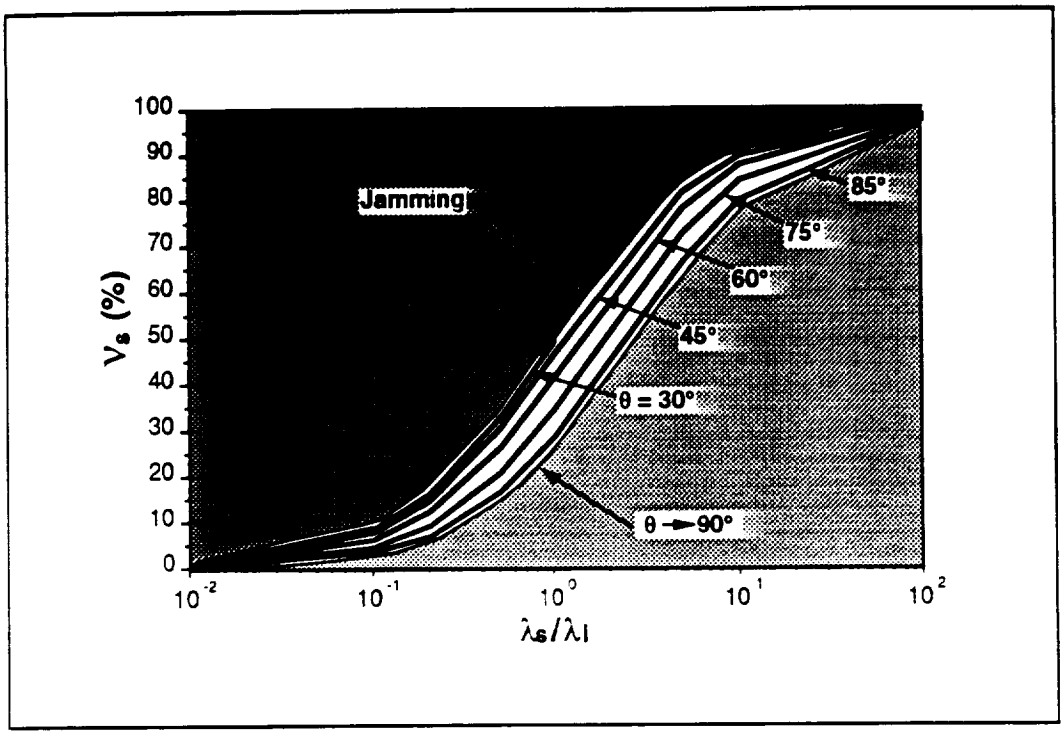


Figure 5.14 Percentage of stitch fibers versus ratio of stitch-to-insertion yarn linear density (tricot stitch, κ = 0.75, ρ = 2.5 kg m-3, fi = 5, and η = 0.5).

5.4 References

5.1. Ko, F. K., 1989, "Three-Dimensional Fabrics for Composites", *Textile Structural Composites*, Vol.3, Elsevier, Amsterdam, the Netherlands, pp. 129-171.

5.2. Pastore, C. M. and Ko, F. K., 1990, "Modeling of Textile Structural Composites, Part I: Processing-science Model or Three-dimensional Braiding", *J. Text. Inst.*, Vol. 81, pp. 480-490.

5.3. Li, W., Hammad, M. and El-Shiekh, A., 1990, "Structural Analysis of 3-D Braided Preforms for Composites, Part I: The Four-step Preforms", *J. Text. Inst.*, Vol. 81, pp. 491-514.

5.4. Yang. J. M., Ma, C. L. and Chou, T. W., 1986, "Fiber Inclination Model of Three-Dimensional Textile Structural Composites", *J. Comp. mat.*, Vol. 20, pp. 472-484.

5.5. Du, G. W., Popper, P. and Chou, T. W., 1991, "Analysis of Three-dimensional Textile Preforms for Multidirectional Reinforcement of Composites", *J. Mat. Sci.*, Vol. 26, 3438-3448.

5.6. Du, G. W., and Ko, F. K., 1991, "Geometric Modeling of 3-D Braided Preforms for Composites", *Proceedings of 5th Textile Structural Composites Symposium*, Drexel University, Philadelphia, PA, Dec. 4-6.

5.7. Du, G. W. and Ko, F. K., "Process Simulation and Control of 3-D Braiding", paper under preparation.

5.8. Cai, Y. J., Du, G. W. and Ko, F. K., 1991, "Solid Modeling of Fiber Architecture", *ACT (Advanced Composite Technology) Mechanics of Textile Composites Workshop*, Hampton, VA, April 2-3.

5.9. Ko, F. K., Bruner, J., Pastore, A., Scardino, F., 1980, "Development of Multi-Bar Weft Insertion Warp Knit Fabric for Industrial Applications", **ASME** Paper No. 90-TEXT-7, October 1980.

5.10. Ko, F. K., Krauland, K., and Scardino, F., 1982, "Weft Insertion Warp Knit for Hybrid Composites", **Proceedings of the Fourth International Conference on Composites**, 1982.

5.11. Ko, F. K., Fang, P., and Pastore, C., 1985, "Multilayer Multidirectional Warp Knit Fabrics for Industrial Applications", **J. Industrial Fabrics**, Vol. 4, No. 2, 1985.

5.12. Ko, F. K., Pastore, C., Yang, J. M., and Chou, T. W., 1986, "Structure and Properties of Multi-layer Multi-directional Warp Knit Fabric Reinforced Composites", **Proceedings of the Third US.-Japan Conference on Composites**, Tokyo, 1986.

5.13. Ko, F. K., and Kutz, J., 1988, "Multiaxial Warp Knit for Advanced Composites", **Proceedings of the Fourth Annual Conference on Advanced Composites**, Dearborn, Michigan, US., Sept. 1988.

Chapter - 6 Applications

6.1 Implementation of the Geometric Model for pre-form design

The geometric model is implemented on a Sun 3/160 work station using C language. The computer code has a modular structure, each module performs a clearly defined function. New modules can be added to increase the capabilities of the code. Current capacity of the code include the computer simulation of the 3D braided structure (3DB) and the Multiaxial Warp Knit (MWK) structure.

The simulation is divided into stages. In stage 1, a group of modules, which were compiled into an executable file named preform **generator**, is executed. The processing and/or geometric parameters are input from keyboard. Based on the logic described in earlier chapters, the yarn path is determined and a device independent, neutral graphics file is written. In stage two a model **depictor** converts this neutral data files into screen images. The Sun graphics library calls are used to draw the graphics primitives. The logic flow of different modules are given below. The stage 1 is device independent. The data files after stage one are written in ASCII text so that the files are transferable from one computer system to the other and all the modules need not to be run on the same machine.

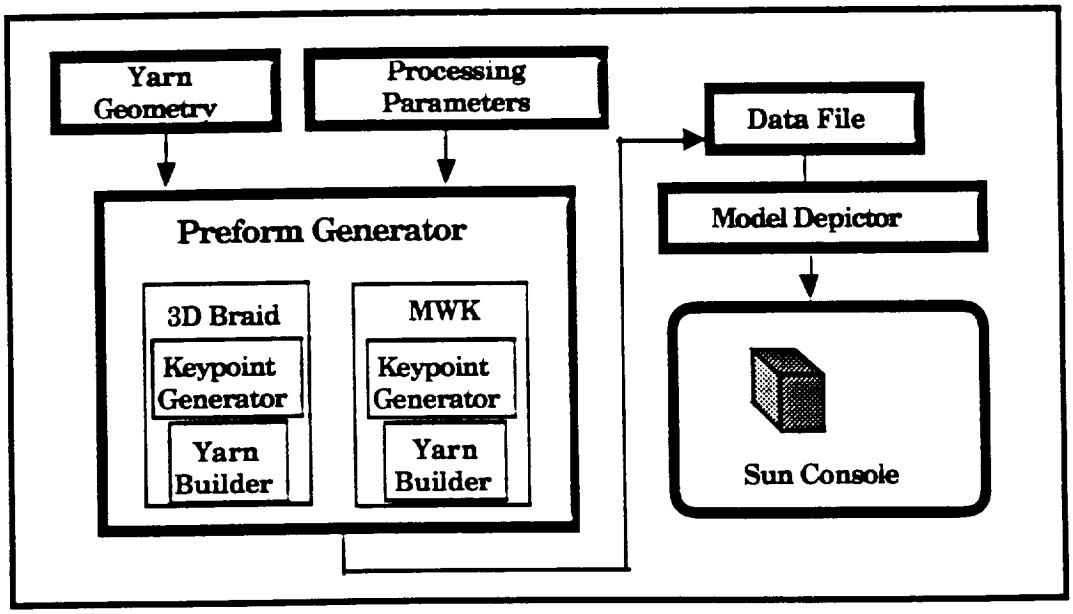


Fig. 6.1 The Simulation Engine

6.1.1 The Preform Generator

Current capabilities for preform generation include the 3D braid and the MWK. The computer codes to perform the preform generation were developed in the C computer language. C is a powerful language and is available in practically all unix based workstations. The codes are portable with minimal effort and hence are hardware independent. The basic principle of preform generation is to identify a number of key points along the yarn trace and then to pass a yarn through the keypoints.

Preform generation for 3D braid

The keypoints for the 3D braided preforms are generated from the machine motion. The yarn trace is determined by taking keypoints along the machine motion and then passing a B-spline along the keypoints. The yarn path is simulated such that it assumes a shape that minimizes the energy of the system. To accomplish this additional keypoints along the yarn center line is generated. These keypoints through which the B-spline passes are called knots in spline vernacular. The details of the yarn path modeling are given in Chapter 3.

Once the yarn trace is established the yarn cross-section is taken as a input and it is swept along the trace to generate the solid yarn. The twist of the yarn at the turning point is calculated. A neutral, device independent graphics file is generated for later rendering by a depiction program.

The keypoint generation for 3-D braided preform is accomplished by **3dbraid**. The default name for the output the intermediate data file is **movement.dat** but it can always be renamed. Once the yarn-path is generated the solid yarn is built along the path by calling the module **build_yarn**. The graphics output of this stage is written in neutral graphics file. The default name for the device independent graphics file is **n.dat** but is usually renamed to represent the geometry it renders.

Preform generation for MWK

The keypoint trace for the MWK structure follows the same principle as 3-D braids, although the actual implementation involves a few more steps. The added complexity arises from the fact that the MWK structure consists of both insertion and stitch yarns that have to be considered separately. The keypoints for the stitch yarns are generated by **mwk**. The keypoints for the insertion yarns are selected by observation. The body around the trace of insertion yarn is generated by the program **xybuild**, the body around the trace of the stitch yarn is built by **build_yarn** (same program as used for the 3D braid). The theory of

selecting the keypoints is given in Chapter 3 and the details of the implementation is given in the user manual section of this report.

6.1.3 The Model Depictor

The neutral graphics data file generated by the model generators **3dbraid** (for 3DB) or **mwk** (for MWK) in conjunction with **build_yarn** and/or **xybuild**, is rendered on the computer screen at this stage. The depiction program used is called **render**. The depiction program is more than just a image renderer, it can also manipulate the image by rotating it for different views and produce section diagrams by passing a cutting plane through the structure.

The rendering program reads in the neutral graphics file and depicts it on the screen. While the rendering logic is similar for all devices the actual subroutine calls are particular to the hardware used. In the current project the hardware used to render the model is a Sun 3/160 workstation. The workstation is equipped with a Graphics Processor (GP). Sun 3/160 is equipped with a 8 bit color display. The value of each pixel serves as an index to a colormap table. This techniques allows the mapping of up to 16 million colors. At anyone time the 8 bit monitor can however display up to 256 colors simultaneously.

The rendering of the neutral graphics file is accomplished by fully utilizing the SUN systems SunView GUI (graphics user interface). SunView is Sun Microsystems proprietary system. It supports interactive graphics based applications running within window environment. The driving engine for the renderer is **render**. The use of the modules **3dbraid**, **mwk**, **build_yarn** and **render** are described in section 6.1.4. The code is listed in Appendix A.

6.1.4 Installation of Software on a SUN workstation.

The software is available on a SUN QIC 24 9 track data cartridge in **tar** format. To install the software on to a SUN workstation the cartridge is to be placed in the drive and the files to be extracted using the following command:

```
tar -xvf /rst0/nasa3d.tar
```

rst0 is the drive name and may vary from system to system. This will automatically create the necessary subdirectories and install the code.

The program is supplied as executables that will run on a SUN 3/160 workstation. To obtain copies of the program contact FMRC at the address on the cover page.

6.1.5 Using the modeling software - User Instructions

Overview

The Simulation takes the following three steps to complete:

Step 1: Depending on which preform structure (either the 3d braid or the Multiaxial warp knit) is to be depicted, the modules **3dbraid** or **mwk** is invoked to generate yarn keypoints. The default name of the intermediate output after this step is the ASCII text file **movement.dat** for 3dbraid and **outfil** for MWK.

Step 2: The solid yarn is built around the yarn trace using the module **build_yarn** and/or **xybuild**. The default name of the output file from this step is **n.dat**.

Step 3: The neutral graphics file generated is then rendered on the screen by the module **render**.

In the rest of the section step by step user instruction to generate yarn keypoints for a simple 3d braided structure and a simple Multiaxial warp knit structure is given. In the following section all the user commands are described in detail for use as a reference.

Preform Generation for 3d braided structure - User Instruction

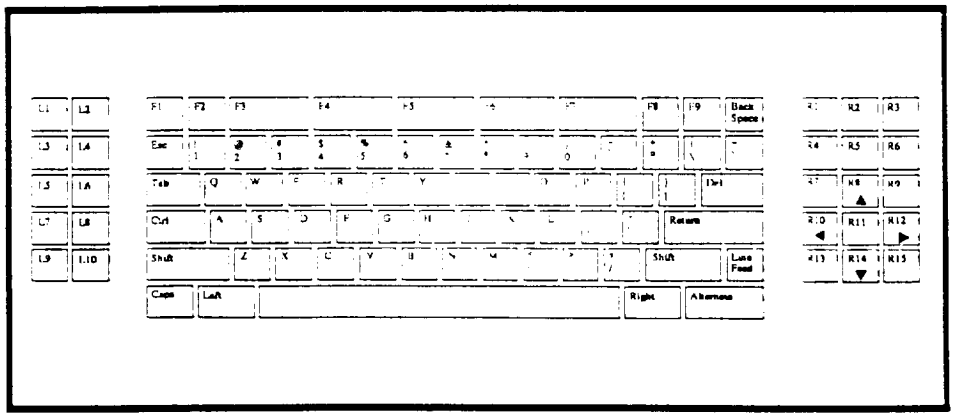


Fig 6.2. The Sun 3/160 Key Board

1. After the software is installed, the model generation is started by typing

3dbraid <Return>

The user will be prompted for the number of tracks and number of column information. The queries and input are shown below. The user input is bolded.

How many tracks? **4** <Return>
How many columns? **5** <Return>

Once the track and column information is input the user is prompted for the number of movements to be simulated. For 4-step braiding, 4 movements complete a cycle. The final prompt is for takeup distance.

How many movement?**2** <Return>

Distance?1 <Enter or Return>

The input is checked for loom size. Currently the maximum allowable loom size is 50x50. If either track or column exceeds the allowable value the input is rejected and the user is prompted again.

Loom too big! # of track or # of column should be less than 50.
Try again!
How many tracks?

The data file generated by this step is named **movement.dat**. The data file for the example inputs shown is appended in Appendix A2. The trace of the yarn path is recorded in **movement.dat** .

2. Once the **movement.dat** is generated , the program **build_yarn** is used to place a solid yarn along the trace. Proper yarn twist at changeover points is also incorporated at this point. The task is started by typing

build_yarn <return>

The user is prompted for the geometry of the yarn. The cross-section is modeled as an ellipse and the user is input for the major axis and the minor axis. For a circular cross-section, the user needs to input the radius twice. An example session is shown below.

build_yarn movement.dat<return>

Yarn radial 1? 0.5<return>

Yarn radial 2? 0.25<return>

The output is a neutral graphics file in text format and is automatically named **n.dat** (included in Appendix A2).

It is important to note that steps 1 and 2 do not necessarily has to be carried on the Sun. The user can run the program at his computer of choice and transfer the output **n.dat** to the Sun for graphical rendering.

Preform Generation for MWK - User Instruction

The preform generation for the MWK follows similar principles as the 3Dbraid.

1. The first step is to generate the keypoints for the stitch yarn. This is done by typing **mwk.** On starting the **mwk** the user is taken through a series of queries for the processing parameter. A sample query is shown below.

mwk <return>

Enter # of insertion yarn/unit length

2 <return>

Enter the angle (degree) of bias yarns

40 <return>

Enter the radius of the insertion yarn

.4 <return>

Enter the radius of the stitch yarn

.125 <return>

Enter the radius of the stitch loop

.2 <return>

color of the stitch yarn: 1, 2, 3, 4

1 <return>

2. Once all the necessary input is given to the computer an intermediate **outfil** is generated. This outfil records all the keypoints along the trace of the stitch yarns.

3. Next step is to build the trace of the insertion yarns. The insertion yarns are all arranged along straight lines and hence the two end points of each yarn are taken as the keypoints. The user needs to know the number of insertion yarns he or she needs to plot and then calculate the coordinates of the end points of the trace. The user then edits a small file, **outfil2**, with all the keypoints. A sample **outfil2**, file is shown below.

```
9 120
1 0
0.0 0.5 0.225
1.0 0.5 0.225
1 0
0.0 0.5 0.225
1.0 0.5 0.225
1 0
0.0 1.5 0.225
1.0 1.5 0.225
2 90
0.0 0.0 0.625
0.0 2.0 0.625
2 90
1.0 0.0 0.625
1.0 2.0 0.625
3 45
0.0 0.0 1.025
1.0 1.0 1.025
3 45
0.0 1.0 1.025
1.0 2.0 1.025
4 -45
1.0 0.0 1.425
0.0 1.0 1.425
4 -45
1.0 1.0 1.425
0.0 2.0 1.425
```

The data is entered in free format. The first line has two elements. The first element denotes the total number of yarns to be plotted the second number denotes the number of sections

each yarn is to have. The more the number of section the smoother is the yarn but at the expense of longer computer time. After the first line there is a three line block of data for each line. The first line in each block denotes the yarn color and the second number the orientation of the yarn. The remaining two lines give the starting and end point coordinates. In the sample file the data is prepared for nine insertion yarns.

4. The body is built now around the yarn traces. The program **build_yarn** is used to build around the stitch yarns. The **build_yarn** is the same program as used to build the 3dbraid yarns. A modified version of **build_yarn**, named the **xybuild** is used to build body around the trace of the insertion yarns.

The programs are started by typing

build_yarn outfil <return>

mv n.dat m.dat

xybuild outfil2 <return>

The user dialog is very similar to that described for build_yarn in the section for 3Dbraid and the user is referred to that section. The default output from the code is **n.dat** . So care should be taken to rename the n.dat generated by build_yarn before xybuild is run or the original n.dat will be overwritten.

Once the two data files **m.dat** and **n.dat** are generated they are combined to one file. Each of the files have two header lines which contain important information that is needed for successful combination. A sample header for **n.dat** is shown below.

9 6462 6480

1.2000 -0.2000 2.1414 -0.1414 1.5982 0.0518

718 720

0.000000 0.700000 0.225000

Each file contains one block of data for each yarn. Each block can be identified by its header line which contains only two integers. In this example file the block header file starts with the integers 718 720. The first block from **m.dat** contains only one block of data. The first block of **n.dat** is replaced by the data from **n.dat**. The second line of **n.dat** is now compared to second line of **m.dat** and modified as explained below. The second line contains the information of the three-dimensional parallelepiped that encompasses the MWK structure. It is defined by inputting the range of x, y and z as shown:

```
xmax   xmin   ymax ymin   zmax zmin
1.2000 -0.2000 2.1414 -0.1414 1.5982 0.0518
```

The xmax, xmin, ymax, ymin, zmax and zmin of the two files are compared and the maximum value for xmax, ymax and zmax and the minimum values for xmin, ymin and zmin are chosen. These values are used to modify the second line of **n.dat**. The neutral graphics file is now ready for rendering.

Graphical Rendering

The graphical rendering program is developed for a sun workstation environment. The program is highly interactive. Initial input is by Keyboard but once the image is drawn on the screen, further manipulation is done by the Sun Mouse.

The optimum display is obtained when the SunView window covers the whole screen. To ensure this the user has two choices.

(a) The user can click at the corner of the active screen and select the **full screen window** option of the popup menu. Some useful SunView window management techniques are discussed in Section 5.

(b) The user can run the rendering program without starting SunView. In this case the rendering program starts SunView and makes the active window span the whole screen.

To start the processing of **n.dat** neutral graphics file, type

```
model n.dat <return>
input cut plane normal 0.1 0.1 0.1 <return>
```

At this point the control buttons for image manipulation pops up on the right side of the active window. The screen is shown in Fig. 6.3.

After the program starts the image is manipulated by the control strip. Different image manipulation functions are performed by placing the cursor at the appropriate button and depressing the middle mouse button. A schematic of the Sun Mouse is shown in Fig. 6.4

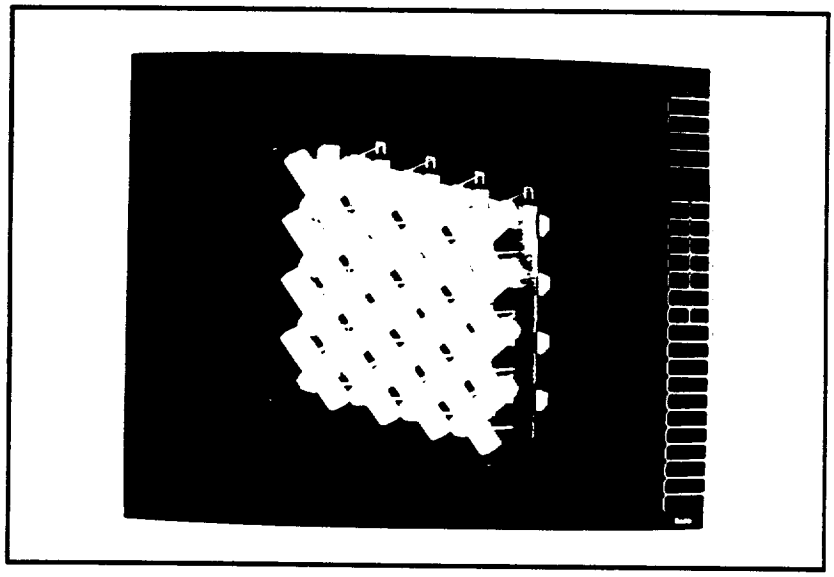


Fig. 6.3 Screen Display during Simulation

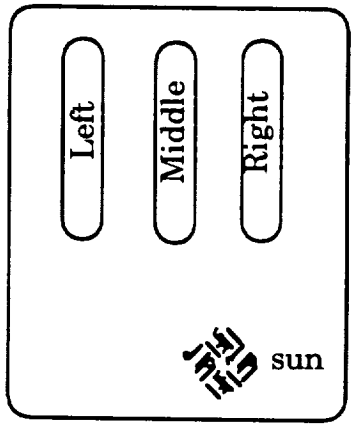


Fig. 6.4 - The sun three button mouse.

In the rendering program only the **left** button is used. The **right** button is used by SunView to change the size of the windows and to exit from SunView. A schematic of the control strip is shown in Fig. 6.5. As can be seen the control strip offers numerous options. All these options are explained in the next section. In this session the minimal steps necessary to render the neutral graphics file is described.

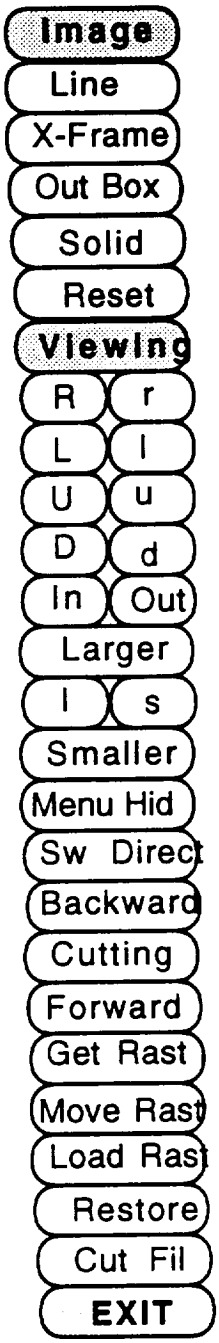


Fig. 6.5 Schematic of the Control Strip

1. The first step is to select a bounding parallelepiped inside which the model will be drawn. The user points and clicks at the option **Outbox.** A wire frame parallelepiped is drawn on the screen.

2. The user then controls the size of the parallelopiped by clicking the **viewing** buttons. The **R, L, U** and **D** buttons move the viewpoint to right, left, up and down respectively. The corresponding lowercase buttons perform the same function but at a slower rate and hence can be used to fine tune the position of the box. The **Larger, Smaller, l** and **s** buttons are used to increase the size of the box. Larger and Smaller buttons increases or decreases the size at a higher rate than the l and s buttons respectively.

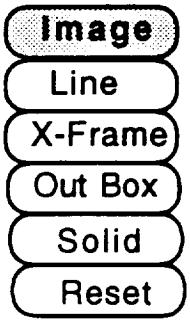
3. After the box is placed the yarn can be drawn by using one of the image buttons. The X-Frame button draws a wire frame image of the yarn. The Solid option draws a hidden line solid model.

4. The image can now be saved as a raster file by using the Move Rast button.

5. To view a cross-section along a cutting plane the user uses the **cutting** in conjunction with the **Forward** and **Backward** buttons. While the cutting plane is predetermined, it can be moved along its normal by the **Forward** and **Backward** buttons. The position of the plane can be seen on the display. Once the cutting plane is positioned, the cross section is displayed by clicking on the **cutting** button.

6.1.5 Command reference of the image manipulation controls

The image manipulation commands are cataloged below. To select (or use) any of the commands the user needs to position the cursor in the appropriate box and press the **left** mouse button.



The Image Buttons

These buttons Control the Type of Image Generated by the code.

Line

Selecting the **Line** button draws a wire frame mesh of the model within the bounding box.

X-frame

This button is not in use at this time.

Outbox

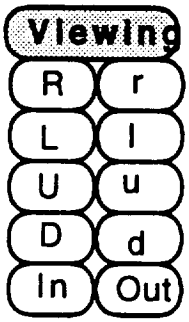
Draws the bounding box for the model.

Solid

Draws a hidden plot model of the yarn.

Reset

Resets the viewing parameters to default and erases the screen.



The Viewing Buttons

The viewing buttons are used to position the outbox to look at the unit cell from different aspects.

R and r

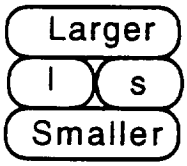
The **R** button is used to rotate the outbox counter clockwise. The **r** button is used to rotate the outbox in the same direction but at a slower rate.

L and l

The **L** button is used to rotate the outbox clockwise. The **l** button is used to rotate the outbox in the same direction but at a slower rate.

U and u

The **U** button is used to move the outbox up in the positive z direction. The **u** button is used to move the outbox in the same direction but at a slower rate.



The Size Buttons

The size buttons control the size of the outbox

Larger

The **Larger** button increases the screen size of the outbox. The effect is to zoom in along the viewing line.

Smaller

The **Smaller** button decreases the screen size of the outbox. The effect is to zoom out along the viewing line.

l and s

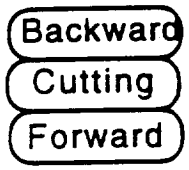
The **l** and **s** buttons are the slow speed counterparts of the **Larger** and **Smaller** buttons.

Menu Hid

The **Menu Hid** button hides the menu. This is useful for taking snapshot of the screen.

Sw Direct

This button is not in use



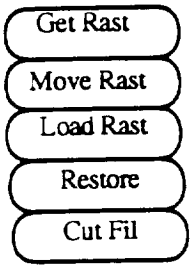
The Cutting Buttons

Backward

The **Backward** button moves the cutting plane along a normal to the plane in the negative direction.

Forward

The **Forward** button moves the cutting plane along a normal to the plane in the positive direction.



The File Buttons

The file buttons manipulates the sun raster files.

Get Rast

The **Get Rast** button reads are raster file and displays it on the screen.

Move Rast

The **Move Rast** button creates a raster file with the current screen image.

Load Rast

The **Load Rast** button displays the contents of a raster file

Restore

The **Restore** button restores the position of the cutting plane

The Exit Button

Selecting the **exit** button quits the program and closes the model window.

6.1.6 Starting the Rendering program inside SunView

The rendering code when started from outside SunView automatically invokes SunView. That is the easiest way to start the code. On the other hand starting the code from inside the code has the advantage that the user can predetermine the size of the window in which the model will be drawn. The user can also have other windows open that he can use in conjunction with running the program. In this section limited instructions are given to manipulate the windows for such a purpose. For exhaustive discussion on the SunView environment the user is referred to [6.1.2].

To start the rendering program place the cursor in a command or shell window. Fig. 6 depicts a typical SunView desktop with a open shell window. The frequently used techniques are described below

Opening a New Shell

New Shell Windows are opened from the SunView menu. SunView menu is opened by clicking the right mouse button when none of the windows are active. From the SunView menu selecting the item **Shells** pops up a new Shell.

Resizing the Shell Window

The size of the shell window determines the portion of the screen that will display the model. To change the size of the shell window the cursor is placed on the top window border. Pressing the right mouse button pops up a Frame Menu. The Resize menu item is used to control the window size. The Resize is a pull right menu. The choices offered by the menu are constrained and unconstrained resizing, zoom and Full Screen. The cursor is then placed near the corner or frame border. Pressing the middle mouse button the bounding box can be adjusted to requirement. Zoom expands the window vertically to maximum and Full Screen expands the window to encompass the whole screen.

Exiting SunView

To exit SunView the SunView menu is popped up and the item Exit SunView is selected.

6.2 Application for Engineering Design

The visualization of the architecture on the computer allows us to look at the model without going through the time consuming and expensive operation of actually making a prototype. Once a textile pre-form is decided upon it can be readily discretized into a finite structure. The software developed can allow us to connect it to a structural analysis program.

6.2.1 Automesh Generation

The graphical rendering of any shape requires the object to be discretized into a finite number of polygons. The larger the number of polygons is, the realistic the depiction is. This discretization can directly be used to obtain a finite element model of the unit cell. The polygons that describe the surfaces are taken as a single element and the triangular elements Each triangle is systematically broken up into four smaller triangles. Adjacent triangles are then combined to form as many quadrilateral elements as possible.

6.2.2 A Case Study

As an example of application of the modeling techniques in engineering design we present a work carried out at the fibrous materials research center. The center was called upon to design and fabricate flanged tube for a helicopter. The flanged tube was designed to transmit torque from engine to the hub of a helicopter blade assembly. The tube was to be made of composite material. It was suggested by the client to use IM7-W 12k graphite fiber and to use PR500 resin as the matrix. The matrix and fiber properties are given in Table 6.2

Properties	IM7-W	PR500
Tensile Strength(Ksi)	769	8.3
Tensile Modulus(Msi)	44	.507
Tensile Strain(%)	1.79	1.9
Density(g/cc)	1.78	1.245
Tg (°C)		205

The flanged tube was subjected to static bending and fatigue loading. The static loading comprised of 771,400 in-lb torque and 38,750 in-lb moment; the fatigue loading was 175,000±175,000 in-lb torque and 0±27,680 in-lb moment. The torque and bending moment occured at the same time.

The fabrication process included first layer 2-D braiding on tubular flange, thickness built-up with woven fabrics and then overbraiding on tube and flange for final process. The braiding angle of the flange tube is ±45°. The fiber volume fraction is about 50%.

Geometric Modeling

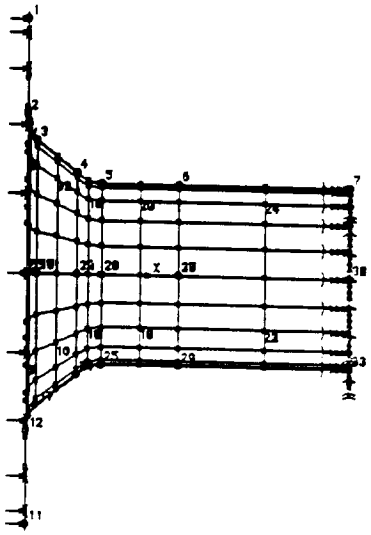
In braiding operations, all braiding yarns move with their carriers and wrap around the surface of the mandrel. One group of braiding yarns moving clockwise forms an angle of $+\theta$ with the mandrel axis; the other group of braiding yarns, moving anticlockwise, has an angle of $-\theta$. These two groups of braiding yarns ($\pm\theta$) are interlocked, forming a biaxial fabric on the mandrel.

Braids can be formed with different yarn interlacing patterns in a manner similar to that of woven fabrics simply by changing relative position of carriers on the track ring. If one bias yarn continuously passes over and under one yarn of the opposing group, the pattern is designated as 1/1 (or diamond) braid. Other simple interlacing patterns in common use include 2/2, 3/3, 2/1 and 3/1 braids. After examing each pattern and considering the resulting volume fraction and effective moduli the 2/2 biaxial braid was chosen for the flanged tube. The effective properties were then used in a finite element model to complete the design process.

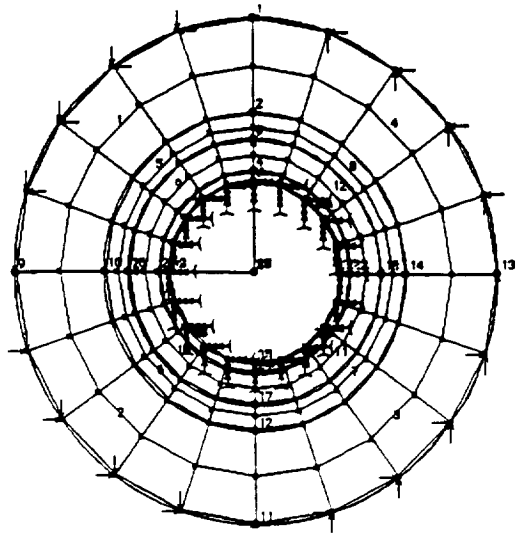
Finite Element Modelling

The finite element analysis(FEA) is performed in order to examine the stress distribution in the flanged tube when it is under the operation environment. The FEA procedures include the definition of the geometry, the boundary conditions, the material properties and element type in the analysis. In this project, 2-D shell element is chosen for the finite element analysis. In the analysis the total number of nodes is 260, and the total number of elements is 240. The created geometry and the element meshes are shown in Figure 6.6 from different views.

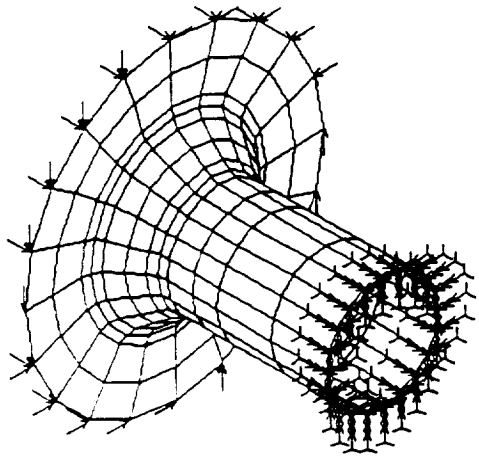
The element type used for the finite element analysis is **nonlinear composite quadrilateral shell element.** This is a 4-node muti-layer quadrilateral shell element with bending consideration, which is capable of analyzing three-dimensional structures. Six degrees of freedom (three translations and three rotations) per node are considered.



view : [001]



view : [100]



view : [111]

Figure 6.6 The geometry and element meshes of the flanged tube.

For the boundary conditions, fixed dispacements at the end of tube are imposed, while the torque and bending moment are applied on the edge of the flange according to the provided loading environment. The loadong conditions are illustrated in Figure 6.7. Both the torque and the bending moment can be equated by a set of force components uniformly distributed over the nodes on the edge of the flange. For the purpose of simplicity, only 9 inches of the tube in length is considered in the analysis.

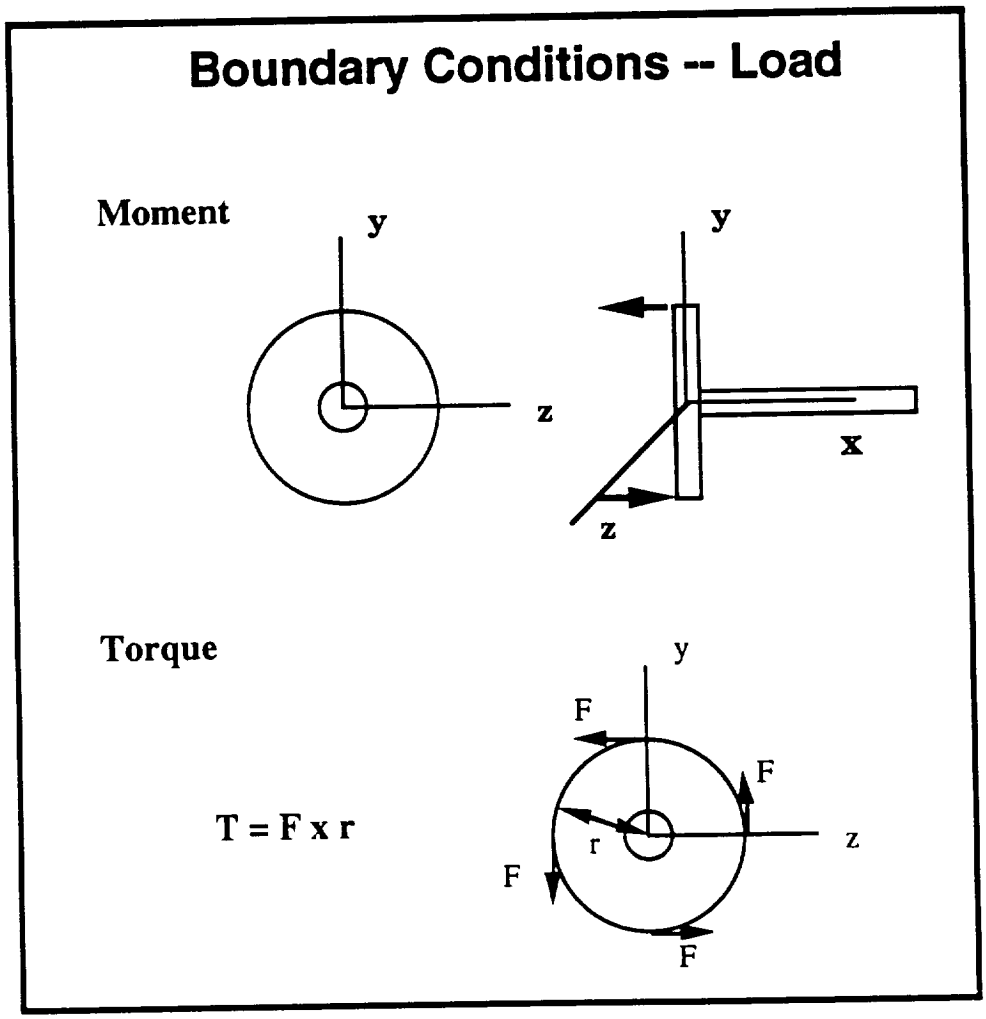


Figure 6.7 The loading boundary conditions of the flanged tube.

The stiffness matrix of the flanged composite tube obtained using the Fabric Geometry Model is given as follows:

$$\begin{bmatrix} 6.15 & 4.88 & 0.27 & 0 & 0 & 0 \\ 4.88 & 6.15 & 0.27 & 0 & 0 & 0 \\ 0.27 & 0.27 & 1.33 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.63 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.63 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5.13 \end{bmatrix} \text{Msi}$$

The stiffness matrix can be used as input into the COSMOS finite element package to compute the stress and displacement distributions of the flanged composite tube. The results, such as σ_x, σ_y, σ_z, τ_{xy}, τ_{xz}, τ_{yz}, von-Mises stress, U_y,U_z and resultant displacement U_{tot}, are used to see if any material or design limit is exceeded. As an example of the von Mises stress fringes are presented in Figures 6.8. From the figures, the maximum stress occurs in the tube portion and the value of the maximum stress does not exceed the material limit.

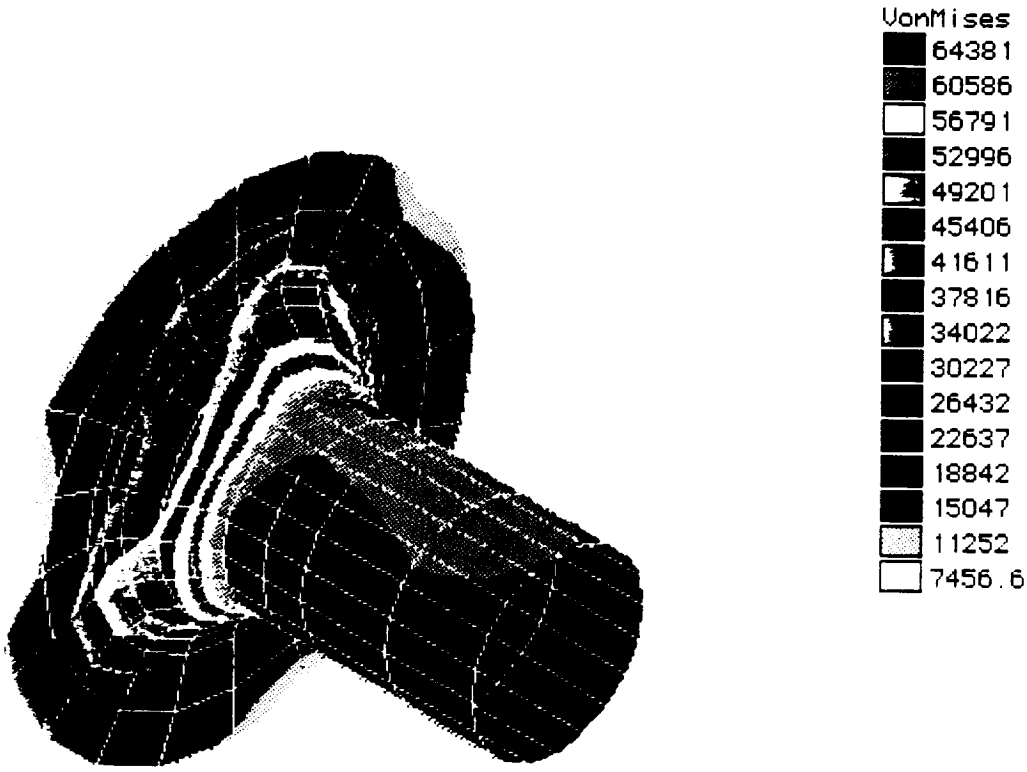


Figure 6.8 The contour plot of von Mises stress in the flanged tube.

Appendices

A1 Program

Model Generators

3 D BRAID - MOVEMENT GENERATOR

```
/* Movement Generator  */

# include <math.h>
#include <usercore.h>
#include <stdio.h>

#define SIZE 50
#define FASLE 0

static short x_matrix[SIZE][SIZE][2],y_matrix[SIZE][SIZE][2],initial[SIZE][SIZE];
static int n_track,n_column;
static int movement,right,up,track,column;
static int yarn_index;
static float dist,yarn_locat[SIZE][3];
int yarn_color,n_pair,n_knot;
short yarn_matrix[SIZE][SIZE][10];
struct {
        float x;
        float y;
        float z;
        }knotxyz[100];
struct {
        short index0;
        short index1;
        short i;
        short j;
        short ii;
        short jj;
        short flag;
        }knot[100];

main()
{
int i,j,k,ii;
FILE *fptr;

again: printf("\nHow many tracks? ");
scanf("%d",&n_track);
printf("\nHow many columns? ");
scanf("%d",&n_column);
printf("\nHow many movement?");
scanf("%d",&movement);
printf("\nDistance?");
scanf("%f",&dist);
yarn_index=n_knot;
if (n_track>SIZE || n_column>SIZE )
        {
        printf("\nLoom too big! # of track or # of column should  be less than 50. \n Try again!");
        goto again;
        }
```

```
build_xy_matrix();
print_xy_matrix();


 yarn_index=0;
for (i=0;i<n_track;i++)
        for(j=0;j<n_column;j++)
        if (y_matrix[i][j][0] !=0 || x_matrix[i][j][0] !=0)
                {
        yarn_index++;
        initial[i][j]=1;
        }
for (i=0;i<n_track;i++)
        {
        for(j=0;j<n_column;j++)
                printf("%d",initial[i][j]);
        printf("\n");
        }

fptr = fopen("movement.dat","w");
/*fprintf(fptr,"%d %d\n",yarn_index,movement/2+1);
*/
fprintf(fptr,"%d %d\n",yarn_index,movement+1);

movement++;
yarn_color = 3;
yarn_index=0;
for (i=0;i<n_track;i++)
        for (j=0;j<n_column;j++)
        {
        if (initial[i][j] !=0)/* if initial[i][j]=1, it is yarn! */
                {
                for ( k=0;k<movement;k++)
                {
                        if (k==0)
                                {
                        track=yarn_locat[k][2]=i+1;
                        yarn_locat[k][1]=0;/* y coord. is ZERO */
                        column=yarn_locat[k][0]=j+1;
                        right=0;
                        up=0;
                                }
                        else
                                {
                                if (k%2 !=0)/* move track first       */
                                        move_track(k);
                                else
                                        move_column(k);
                                }
                        yarn_locat[k][1]= k*dist ;
                }       /* end of movement */

                printf("yarn # %d\n",yarn_index);

                fprintf(fptr,"%d\n",yarn_color);
                yarn_color= (yarn_color==3) ? 4 : 3;
```

```
                for (k=0;k<movement;k++)
                        /*if (k%2==0)*/
                                                {
                printf("%2.4f %2.4f %2.4f\n",yarn_locat[k][0],yarn_locat[k][1],yarn_locat[k][2]);
                fprintf(fptr," %2.1f                    %2.1f
%2.1f\n",yarn_locat[k][0],yarn_locat[k][1],yarn_locat[k][2]*0.7);
                        }
                fprintf(fptr,"\n");
                yarn_index++;
                }

        } /* end of yarn location   */
initial_yarn_matrix();
i=0;
for (k=0;k<movement-1;k++){
                if(k%2) shift_track(i,i+1);
                else    shift_column(i,i+1);
        if(k%2 && k !=0 ){
                n_pair = 0;
                knot_search(i-1,i+1);
                pick_knot(n_pair,k*dist);
        }
        i++;
}

for (i=0;i<n_knot;i++)
        printf("\n%2.2f  %2.2f    %2.2f    ",knotxyz[i].x,knotxyz[i].y,knotxyz[i].z);
fprintf(fptr,"%d\n",n_knot);
for (i=0;i<n_knot;i++)
        fprintf(fptr,"\n%2.2f      %2.2f    %2.2f    ",
        knotxyz[i].x,knotxyz[i].y,knotxyz[i].z);
close (fptr);
}       /* end of main program   */

move_track(k) int k;
{
        if (right==0)
                {/* move right    */
                column=yarn_locat[k][0]=column+x_matrix[track-1][column-1][right];
                yarn_locat[k][2]=yarn_locat[k-1][2];/* keep Z direction */


                right=1;
                }
        else/* move left   */
                {column=yarn_locat[k][0]=column+x_matrix[track-1][column-1][right];
                yarn_locat[k][2]=yarn_locat[k-1][2];/* keep Z direction */

                right=0;
                }

} /* end of move_track     */



move_column(k) int k;
{
        if (up==FALSE)
```

```
                {
                track=yarn_locat[k][2]=track+y_matrix[track-1][column-1][up];
                yarn_locat[k][0]=yarn_locat[k-1][0];
                                up=1;
                }
        else
                {
                track=yarn_locat[k][2]=track+y_matrix[track-1][column-1][up];
                yarn_locat[k][0]=yarn_locat[k-1][0];


                up=0;
                }
}       /* end of move_column   */


print_xy_matrix()
{
int i,j,k;

printf("\n");
for (k=0;k<2;k++)
        for(i=0;i<n_track;i++)
        {
        if (k==0)
                {
                for(j=0;j<n_column;j++)
                        printf("%2d ",x_matrix[i][j][0]);
                printf("\n");
                }
        else
                {
                if (k==1 && i==0) printf("\n");
                for(j=0;j<n_column;j++)
                        printf("%2d ",x_matrix[i][j][1]);
                printf("\n");
                }
        }

printf("\n");
for (k=0;k<2;k++)
        for(i=0;i<n_track;i++)
        {
        if (k==0)
                {
                for(j=0;j<n_column;j++)
                        printf("%2d ",y_matrix[i][j][0]);
                printf("\n");
                }
        else
                {
                if (k==1 && i==0) printf("\n");
                for(j=0;j<n_column;j++)
                        printf("%2d ",y_matrix[i][j][1]);
                printf("\n");
                }
        }
}       /* end of print x_y-matrix          */
```

```
build_xy_matrix()
{
int i,j,k;

for (i=0;i<n_track;i++)
        for(j=0;j<n_column;j++)
                for (k=0;k<2;k++)
                        x_matrix[i][j][k]=y_matrix[i][j][k]=initial[i][j]=0;

for(k=0;k<2;k++)
        for(i=0;i<n_track;i++)
        {
                if (k==0)
                {
                if(i !=0 && i !=n_track-1 && i % 2 !=0 )
                        for(j=0;j<n_column-1;j++)
                                x_matrix[i][j][0]= 1;
                if (i !=0 && i !=n_track-1 && i % 2 !=1 )
                        for (j=1;j<n_column;j++)
                                x_matrix[i][j][0]= -1;
                }
                else
                {
                if(i !=0 && i !=n_track-1 && i % 2 !=0 )
                        for(j=1;j<n_column;j++)
                                x_matrix[i][j][1]= -1;
                if (i !=0 && i !=n_track-1 && i % 2 !=1 )
                        for (j=0;j<n_column-1;j++)
                                x_matrix[i][j][1]= 1;
                }
        } /* end of build x_matrix          */


for(k=0;k<2;k++)
        for(i=0;i<n_column;i++)
        {
                if (k==0)
                {
                if(i !=0 && i !=n_column-1 && i % 2 !=0 )
                        for(j=0;j<n_track-1;j++)
                                y_matrix[j][i][0]= 1;
                if (i !=0 && i !=n_column-1 && i % 2 !=1 )
                        for (j=1;j<n_track;j++)
                                y_matrix[j][i][0]= -1;
                }
                else
                {
                if(i !=0 && i !=n_column-1 && i % 2 !=0 )
                        for(j=1;j<n_track;j++)
                                y_matrix[j][i][1]= -1;
                if (i !=0 && i !=n_column-1 && i % 2 !=1 )
                        for (j=0;j<n_track-1;j++)
                                y_matrix[j][i][1]= 1;

                }
        } /* end of build y_matrix          */
```

```
}       /* end of build xy_matrix          */


initial_yarn_matrix()
{int i,j,yarn_index;
yarn_index = 1;
for (i=0;i<n_track;i++)
        for (j=0;j<n_column;j++)
                yarn_matrix[i][j][0] = (initial[i][j]==1) ? yarn_index++ :0;
} /* end of initial_yarn_matrix */

shift_track(index0,index1)int index0,index1;
{int i,j;
for (i=0;i<n_track;i++)
        for (j=0;j<n_column;j++) yarn_matrix[i][j][index1] =
                        yarn_matrix[i][j][index0] ;

for (i=1;i<n_track-1;i++){
        if (yarn_matrix[i][0][index1]>0){/*  RIGHT */
                        for (j=1;j<n_column;j++)
                                yarn_matrix[i][n_column-j][index1] =
                                        yarn_matrix[i][n_column-j-1][index1];
                        yarn_matrix[i][0][index1] = 0;
                }
        else {/*  LEFT */
                        for (j=0;j<n_column-1;j++)
                                yarn_matrix[i][j][index1] =
                                        yarn_matrix[i][j+1][index1];
                        yarn_matrix[i][n_column-1][index1]= 0;
                }

        }/* the next track */
} /* end of shift track */

shift_column(index0,index1)int index0,index1;
{int i,j;
for (i=0;i<n_track;i++)
        for (j=0;j<n_column;j++) yarn_matrix[i][j][index1] =
                                                        yarn_matrix[i][j][index0] ;

for (i=1;i<n_column-1;i++){
        if (yarn_matrix[0][i][index1]>0){/*   DOWN */
                        for (j=1;j<n_track;j++)
                                yarn_matrix[n_track-j][i][index1] =
                                        yarn_matrix[n_track-j-1][i][index1];
                yarn_matrix[0][i][index1] = 0;
                                        }
        else {/*  UP */
                        for (j=0;j<n_track-1;j++)
                                yarn_matrix[j][i][index1] =
                                        yarn_matrix[j+1][i][index1];
                        yarn_matrix[n_track-1][i][index1]= 0;
                }

        }/* the next column */
} /* end of shift column */
```

```
knot_search(index0,index1)int index0,index1;
{
int i,j,ii,jj;
for (i=0;i<n_track;i++)
        for (j=0;j<n_column;j++){
                if (yarn_matrix[i][j][index1] !=0 ){
                        for (ii=0;ii<n_track;ii++)
                                for (jj=0;jj<n_column;jj++){
                                 if (yarn_matrix[i][j][index1]==yarn_matrix[ii][jj][index0] &&
                                        yarn_matrix[ii][jj][index1]==yarn_matrix[i][j][index0] &&
                                        yarn_matrix[ii][jj][index1] != 0){
                                                /*printf("\nPAIR %d
        %d",yarn_matrix[i][j][index1],
                                                yarn_matrix[ii][jj][index1]);*/
                                                knot[n_pair].index0 = yarn_matrix[i][j][index1];
                                                knot[n_pair].index1 = yarn_matrix[ii][jj][index1];
                                                knot[n_pair].i =i;
                                                knot[n_pair].j = j;
                                                knot[n_pair].ii = ii;
                                                knot[n_pair].flag = -1;
                                                knot[n_pair++].jj =jj;
                                                }
                                        }
                        }
        }
}/* end of knot search */


pick_knot(n_pair,y)int n_pair;float y;
{int i,j,k;
for (i=0;i<n_pair;i++){
        if (knot[i].flag == -1){
                knot[i].flag = 1;
                for(j=0;j<n_pair;j++)
                        if ((knot[i].index0 ==knot[j].index0 && knot[j].flag != 1 &&
                                knot[i].index1==knot[j].index1) ||
                                (knot[i].index0 == knot[j].index1 && knot[j].flag != 1 &&
                                knot[i].index1 == knot[j].index0)) knot[j].flag = 0;
                }
        }
for (i=0;i<n_pair;i++)
        if (knot[i].flag == 1){
                knotxyz[n_knot].z = (knot[i].i + knot[i].ii+2)/2.0;
                knotxyz[n_knot].x = (knot[i].j + knot[i].jj+2)/2.0;
                knotxyz[n_knot++].y = y;
                }
}
```

MWK - KEYPOINT GENERATOR

```
/*
  mwk unit cell builder
  history:
        original development in Fortran -> Dr. Charles Lei
        C conversion                Anisur Rahman       4/16/93
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define LOOP 10
#define POINTS_PER_STITCH 33
#define DIMENSIONS 3

int ncolor;
int ind1, ind2, ind3;

float din;

double pi,dx,dy,sqrt2,theta;
double rstitch, rinsert, rloop;
double coor[LOOP][POINTS_PER_STITCH][DIMENSIONS];

FILE *fp;

main(){

  puts (" Enter # of insertion yarn/unit length");
  scanf("%f",&din);
  dx=1./din;
  puts (" Enter the angle (degree) of bias yarns ");
  scanf("%lg",&theta);
  pi = 4.* atan(1.);
  theta=theta/180.*pi;
  dy = dx * tan(pi/2. - theta);
  puts(" Enter the radius of the insertion yarn");
  scanf("%lg",&rinsert);
  puts(" Enter the radius of the stitch yarn");
  scanf("%lg",&rstitch);
  puts(" Enter the radius of the stitch loop");
  scanf("%lg",&rloop);
/*
  Compute the geometry within an unit cell
*/
  for (ind1 = 0;ind1 != LOOP; ind1++)
    for (ind2 =0; ind2 != POINTS_PER_STITCH; ind2++)
      for (ind3=0; ind3 != DIMENSIONS; ind3++)
        coor[ind1][ind2][ind3]=0;
  coor[0][0][0]=dx/2.-rstitch;
  coor[0][0][1]=-rstitch - rloop;
  coor[0][1][0]=dx/2.-4.*rstitch*(2.*rstitch/dy)-rstitch;
  coor[0][1][2]=-2.*rstitch-(2.*rstitch/dy)*4.*rstitch;
  coor[0][2][0]=dx/2.-rloop;
  coor[0][2][1]=dy-rloop-rstitch;
  sqrt2=sqrt(2.);
  coor[0][3][0]=dx/2-rloop/sqrt2;
  coor[0][3][1]=coor[0][2][1]+rloop/sqrt2;
  coor[0][4][0]=dx/2;
  coor[0][4][1]=dy-rstitch;
  coor[0][5][0]=dx/2.+rloop/sqrt2;
  coor[0][5][1]=coor[0][3][1];
  coor[0][6][0]=dx/2.+rloop;
  coor[0][6][1]=coor[0][4][1]-rloop;
```

```
  coor[0][7][0]=dx/2.+rstitch+2.*rstitch*(2.*rstitch/dy);
  coor[0][7][1]=-rstitch;
  coor[0][7][2]=-2.*rstitch;
  coor[0][8][0]=coor[0][4][0]+rstitch;
  coor[0][8][1]=-rloop-rstitch;
  coor[0][9][0]=dx/2.;
  coor[0][9][1]=dy/2.-rinsert*sqrt2;
  coor[0][9][2]=7.*rinsert;
  coor[0][10][0]=dx/2.;
  coor[0][10][1]=dy/2.;
  coor[0][10][2]=8.*rinsert+rstitch;
  coor[0][11][0]=dx/2.;
  coor[0][11][1]=dy/2.+sqrt2*rinsert;
  coor[0][11][2]=7.*rinsert;
  coor[0][12][0]=coor[0][8][0];
  coor[0][12][1]=coor[0][6][1];
  coor[0][13][0]=coor[0][7][0];
  coor[0][13][1]=coor[0][7][1]+dy;
  coor[0][13][2]=-2.*rstitch;
  coor[0][14][0]=coor[0][6][0];
  coor[0][14][1]=coor[0][6][1]+dy;
  coor[0][14][2]=coor[0][6][2];
  coor[0][15][0]=coor[0][5][0];
  coor[0][15][1]=coor[0][5][1]+dy;
  coor[0][16][0]=coor[0][4][0];
  coor[0][16][1]=coor[0][4][1]+dy;
  coor[0][17][0]=coor[0][3][0];
  coor[0][17][1]=coor[0][3][1]+dy;
  coor[0][18][0]=coor[0][2][0];
  coor[0][18][1]=coor[0][2][1]+dy;
  coor[0][19][0]=dx/2.-rstitch-2.*rstitch*(2.*rstitch/dy);
  coor[0][19][1]=dy-rstitch;
  coor[0][19][2]=-2.*rstitch;
  coor[0][20][0]=dx/2.-rstitch;
  coor[0][20][1]=dy-rstitch-rloop;
  coor[0][21][0]=coor[0][9][0];
  coor[0][21][1]=coor[0][9][1]+dy;
  coor[0][21][2]=coor[0][9][2];
  coor[0][22][0]=coor[0][10][0];
  coor[0][22][1]=coor[0][10][1]+dy;
  coor[0][22][2]=coor[0][10][2];
  coor[0][23][0]=coor[0][11][0];
  coor[0][23][1]=coor[0][11][1]+dy;
  coor[0][23][2]=coor[0][11][2];
  coor[0][24][0]=coor[0][20][0];
  coor[0][24][1]=coor[0][20][1]+dy;
  coor[0][24][2]=coor[0][20][2];
  coor[0][25][0]=coor[0][19][0];
  coor[0][25][1]=coor[0][19][1]+dy;
  coor[0][25][2]=coor[0][19][2];
  coor[0][26][0]=coor[0][1][0];
  coor[0][26][1]=coor[0][1][1]+dy*2;
  coor[0][26][2]=coor[0][1][2];
/*
  Write Sun File
*/
  fp = fopen("outfil","w");
  fprintf(fp,"1 27\n");
```

```
  puts(" color of the stitch yarn: 1, 2, 3, 4 \n");
  scanf("%d",&ncolor);
  fprintf(fp,"%d\n",ncolor);
  for(ind1 = 0; ind1 != 27; ind1++)
    fprintf(fp,"%10.6f %10.6f %10.6f\n", coor[0][ind1][0],
                        coor[0][ind1][1],coor[0][ind1][2]);
}
```

3D BRAID YARN BUILDER

```
#include <math.h>
#include <usercore.h>
#include <stdio.h>

#define NC 50
#define NSET 6
#define MAXLINE 400
#define MAXVLINE        5000
#define PI 3.141596

 float cp[NC][3];
 float cm[NC*2][3];

 float bbox[3][2];
 int pi,pj,M,N;

double a[3][3];
 float radial1,radial2,u,v,x,y,z;
 float pp[60000][3];
 int index[60000][6],ind;
 int count ,nvert,npoly,nzset,nvpzset,count1,total_vert,old_count;
 int poly_index,finish_yarn,movement,yarn_color;
 int knotk,knotn,nc;
  float line_vert[MAXVLINE][3];
 int line_index[MAXLINE][8],nline,nlvert;
static float cube[8][3]={{0.0,0.0,0.0},
                                                {0.0,0.0,1.0},
                                                {1.0,0.0,1.0},
                                                {1.0,0.0,0.0},
                                                {0.0,1.5,0.0},
                                                {0.0,1.5,1.0},
                                                {1.0,1.5,1.0},
                                                {1.0,1.5,0.0}};
static float pmatrix[4][4]={{-0.5,1.5,-1.5,0.5},
                                                        {1.0,-2.5,2.0,-0.5},
                                                        {-0.5,0.0,0.5,0.0},
                                                        {0.0,1.0,0.0,0.0}};

FILE *fptr;
struct v_p_count{
        short v_count;
        short p_count;
        } v_p_count[600];

main(argc, argv)
int argc;
char *argv[];
{
```

```
int i,j,ii,ncube;

int k,n,tmp[7],color;
int n_poly_pobj,n_vert_pobj,vert_count,poly_count;
printf("\nYarn radial 1? ");
scanf("%f",&radial1);
printf("\nYarn radial 2? ");
scanf("%f",&radial2);

poly_index = total_vert  = old_count = 0;
finish_yarn = 0;
fptr = fopen(argv[1],"r");
fscanf(fptr ,"%d",&movement);
fscanf(fptr,"%d",&nc);


for (finish_yarn =0;finish_yarn<movement;finish_yarn++)
        {
                fscanf(fptr,"%d",&yarn_color);
        for (i=0; i<nc;i++)
                fscanf(fptr,"%f %f %f",&cp[i][0],&cp[i][1],&cp[i][2]);

        ind=0;

        /*      printf("#%d",finish_yarn);          */

        /*build_control_mesh(nc);*/
        v_p_count[finish_yarn].v_count=v_p_count[finish_yarn].p_count = 0;
        build_yarn();

        }
        /*
cell();
*/
close (fptr);
fptr = fopen ("n.dat","w");


for (i=0;i<total_vert;i++)
        {
        if (i == 0)
                {
                bbox[0][0]=bbox[0][1]=pp[i][0];
                bbox[1][0]=bbox[1][1]=pp[i][1];
                bbox[2][0]=bbox[2][1]=pp[i][2];
                }
        else
                {
                if (bbox[0][0]<pp[i][0]) bbox[0][0] = pp[i][0];
                if (bbox[0][1]>pp[i][0]) bbox[0][1] = pp[i][0];
                if (bbox[1][0]<pp[i][1]) bbox[1][0] = pp[i][1];
                if (bbox[1][1]>pp[i][1]) bbox[1][1] = pp[i][1];
                if (bbox[2][0]<pp[i][2]) bbox[2][0] = pp[i][2];
                if (bbox[2][1]>pp[i][2]) bbox[2][1] = pp[i][2];
                }
        }
```

```
        fprintf(fptr,"%d %d %d\n",movement,poly_index,total_vert);
        fprintf(fptr,"%4.4f      %4.4f      %4.4f      %4.4f      %4.4f
%4.4f\n",bbox[0][0],bbox[0][1],bbox[1][0],bbox[1][1],bbox[2][0],bbox[2][1]);

poly_count = vert_count = 0;
n_poly_pobj = poly_index/movement;
n_vert_pobj = total_vert/movement;

for (i=0;i<        movement;i++){
        n_poly_pobj=v_p_count[i].p_count;
        n_vert_pobj=v_p_count[i].v_count;
        fprintf(fptr,"%d %d \n",n_poly_pobj,n_vert_pobj);
                for ( j=0;j<n_vert_pobj;j++){
                fprintf(fptr,"%f %f %f\n",pp[vert_count][0],
                                pp[vert_count][1],pp[vert_count][2]);
                vert_count++;
                }


        for ( j=0;j<n_poly_pobj;j++){
                fprintf(fptr,"%d %d %d %d %d %d\n",index[poly_count][0],
                index[poly_count][1],index[poly_count][2],index[poly_count][3]
                ,index[poly_count][4],index[poly_count][5]);
                poly_count++;
                }
        }
i = j = 0;
fprintf(fptr,"%d %d \n",i,j);
        close (fptr );

} /* end of main   */



cell()
{
int i,j,k,ii,ncube,color;
int tmp[7];
float x,y,z;


                fscanf(fptr,"%d%d",&ncube,&color);

        if (8*ncube>MAXVLINE)
                                {
                                printf("\n Too many vertices on lines!!");
                                exit(1);
                                }
        nlvert=0;
        for (i=0;i<ncube;i++)
                {
                fscanf(fptr,"%f %f %f",&x,&y,&z);
                for (j=0;j<8;j++)
                        {
                        line_vert[nlvert][0]=cube[j][0]+x;
                        line_vert[nlvert][1]=cube[j][1]+y;
                        line_vert[nlvert][2]=cube[j][2]+z;
```

```
                        nlvert++;

                        }

                }

        tmp[0]=5;tmp[1]=color;
        for (i=0;i<ncube;i++)
                {
                k=i*8+5;
                for (j=1;j<4;j++)
                        {
                        tmp[2]=j+i*8;
                        tmp[3]=k;
                        tmp[4]=k+1;
                        tmp[5]=tmp[2]+1;
                        tmp[6]=tmp[2];
                        k++;

                        for (ii=0;ii<7;ii++)
                                line_index[nline][ii]=tmp[ii];
                        nline++;
                        }
                        tmp[2]=i*8+4;
                        tmp[3]=tmp[2]+4;
                        tmp[4]=tmp[3]-3;
                        tmp[5]=tmp[2]-3;
                        tmp[6]=tmp[2];
                        for (ii=0;ii<7;ii++)
                                line_index[nline][ii]=tmp[ii];
                        nline++;
                        if (nline>MAXLINE)
                                {
                                printf("\n Too many  lines!!");
                                exit(2);
                                }
                }
}       /* end of cell     */

build_yarn()
{
        int k,n,i,j,ii;
        int tmp[6];
        int p_count;
        float ppp[10000][3];
        float tangent[3],vect01[3],vect12[3],b_norm[3],vectpp[3],vect_temp[3],bt[3],
        vect_temp0[3],vect_temp1[3];
        float  dot_product();
        float bn[3];
        float bend,alpha,p[3],angle1,f,pv[3],twist;
        int flag,num,go_back;
        float vtt[3],vv[3],vp[3],vect00[3],vt[3],twist_angle,new_bt[3],new_bn[3];
        float old_b_norm[3],old_bt[3],twist_angle_i;

        struct path
                {
                float bt[3];
                int     node;
```

```
                int twist_flag;
                }path[200];
        k=3     ;
        n=ind-1;
        p_count=0;
        flag = TRUE;
/*
        printf("\nOLD POINTS\n");
        for (i=0;i<=n;i++)
                printf("%4.4f %4.4f %4.4f\n",cm[i][0],cm[i][1],cm[i][2]);
*/
        for (j=0;j<=nc-4;j++){
                for (u=0.0;u<=.9001;u=u+0.2)
                        {
                        cardsp(u,j);
                        ppp[p_count][0]=x;ppp[p_count][1]=y;ppp[p_count][2]=z;
                        p_count++;
                }
        }
        printf("\nNEW POINTS\n");

        j=0;
        for (i=0;i<p_count;i++)
                {
                path[i].twist_flag=FALSE;
                if (fabs(cm[j][1]-ppp[i][1])<=0.05)
                        {
                        path[i].node=TRUE;
                        j++j++;
                        }
                else
                        path[i].node=FALSE;
                }
/*
        for (i=0;i<p_count;i++)
                printf("%4.4f %4.4f %4.4f\n",ppp[i][0],ppp[i][1],ppp[i][2]);
*/

        count =0;

angle1=0.0;
flag = 1;
for (i=0;i<p_count;i++)
        {
printf(" %d\n",i);
if (i==0 )          /* no bending and twisting at the first  points          */
        {
        bend=0.0;
        twist=0.0;
        twist_angle = 0.0;
        for (j=0;j<3;j++)
                        {
                        vect00[j]=ppp[i+1][j]-ppp[i][j];
                        vect01[j]=ppp[i+2][j]-ppp[i][j];
                        }

        cross_product(&bt[0],&vect00[0],&vect01[0]);
        u=dot_product(&vect00[0],&vect01[0]);
```

```
        printf("%2.4f \n",u);
        cross_product(&b_norm[0],&bt[0],&vect00[0]);
        normalize(&bt[0]);
        normalize(&b_norm[0]);
        for (j=0;j<NSET;j++)
                        {
                        alpha= twist_angle +  6.283*j/NSET ;
pp[total_vert][0]=ppp[0][0]+radial1*cos(alpha)*bt[0]+radial2*sin(alpha)*b_norm[0];
pp[total_vert][1]=ppp[0][1]+radial1*cos(alpha)*bt[1]+radial2*sin(alpha)*b_norm[1];
pp[total_vert][2]=ppp[0][2]+radial1*cos(alpha)*bt[2]+radial2*sin(alpha)*b_norm[2];
                        total_vert++;
                        count++;
                        }

                }       /* end of i== 0     */
else
        if ( i == p_count-1)        /* the last one as same as i-1 point  */
                {
                for (j=0;j<3;j++)
                        {
                        vect00[j]=ppp[i-1][j]-ppp[i-2][j];
                        vect01[j]=ppp[i][j]-ppp[i-1][j];
                        }
        /*cross_product(&bt[0],&vect00[0],&vect01[0]);
        u=dot_product(&vect00[0],&vect01[0]);
        printf("%2.4f \n",u);
        cross_product(&b_norm[0],&bt[0],&vect01[0]);

        normalize(&bt[0]);
        */
        bt[0] = old_bt[0];bt[1] = old_bt[1];bt[2] = old_bt[2];
        cross_product(&b_norm[0],&bt[0],&vect01[0]);
        normalize(&b_norm[0]);
                for (j=0;j<NSET;j++)
                        {
                        alpha= twist_angle +  6.283*j/NSET ;
 pp[total_vert][0]=ppp[i][0]+radial1*cos(alpha)*bt[0]+radial2*sin(alpha)*b_norm[0];
 pp[total_vert][1]=ppp[i][1]+radial1*cos(alpha)*bt[1]+radial2*sin(alpha)*b_norm[1];
 pp[total_vert][2]=ppp[i][2]+radial1*cos(alpha)*bt[2]+radial2*sin(alpha)*b_norm[2];
                        total_vert++;
                        count++;
                        }

                }       /* end of i==p_count-1     */
 else
 if ( i==1)
        {
        for (j=0;j<3;j++)
                {
                vect00[j]=ppp[i][j]-ppp[i-1][j];
                vect01[j]=ppp[i+1][j]-ppp[i][j];
                tangent[j]=vect01[j]+vect00[j];
                }
        cross_product(&bt[0],&vect00[0],&vect01[0]);
        u=dot_product(&vect00[0],&vect01[0]);
        printf("%2.4f \n",u);

        cross_product(&b_norm[0],&bt[0],&tangent[0]);
```

```
        normalize(&bt[0]);
        normalize(&b_norm[0]);
        for (j=0;j<NSET;j++)
                        {
                        alpha= twist_angle + 6.283*j/NSET ;

        pp[total_vert][0]=ppp[i][0]+radial1*cos(alpha)*bt[0]+radial2*sin(alpha)*b_norm[0];
pp[total_vert][1]=ppp[i][1]+radial1*cos(alpha)*bt[1]+radial2*sin(alpha)*b_norm[1];
pp[total_vert][2]=ppp[i][2]+radial1*cos(alpha)*bt[2]+radial2*sin(alpha)*b_norm[2];

                        total_vert++;
                        count++;
                        }
        } /* end of i=1 case */
else
if ( i==p_count-2)/* start p-2 case */
        {
        for (j=0;j<3;j++)
                {
                vect00[j]=ppp[i][j]-ppp[i-1][j];
                vect01[j]=ppp[i+1][j]-ppp[i][j];
                tangent[j]=vect01[j]+vect00[j];
                }
        /*cross_product(&bt[0],&vect00[0],&vect01[0]);*/
        bt[0] = old_bt[0];bt[1] = old_bt[1];bt[2] = old_bt[2];

        u=dot_product(&vect00[0],&vect01[0]);
        printf("%2.4f \n",u);

        cross_product(&b_norm[0],&bt[0],&tangent[0]);
        normalize(&bt[0]);
        normalize(&b_norm[0]);
        for (j=0;j<NSET;j++)
                        {
                        alpha= twist_angle + 6.283*j/NSET ;

        pp[total_vert][0]=ppp[i][0]+radial1*cos(alpha)*bt[0]+radial2*sin(alpha)*b_norm[0];
pp[total_vert][1]=ppp[i][1]+radial1*cos(alpha)*bt[1]+radial2*sin(alpha)*b_norm[1];
pp[total_vert][2]=ppp[i][2]+radial1*cos(alpha)*bt[2]+radial2*sin(alpha)*b_norm[2];

                        total_vert++;
                        count++;
                        }
        } /* end of i=p-2 case */
else
        {/*  start the other case */
        for (j=0;j<3;j++)
                {
                vect00[j]=ppp[i-1][j]-ppp[i-2][j];
                vect01[j]=ppp[i][j]-ppp[i-1][j];
                vect12[j]=ppp[i+1][j]-ppp[i][j];
                tangent[j]=vect01[j]+vect12[j];
                }
        cross_product(&vect_temp0[0],&vect01[0],&vect00[0]);
        cross_product(&vect_temp1[0],&vect12[0],&vect01[0]);
        twist= dot_product(&vect_temp0[0],&vect_temp1[0]);
        if (twist >= 0.99 ) twist = 1.0;
        if (twist <= -0.99 ) twist = -1.0;
```

```
        if ( twist == 1.0 || twist == -1.0 )   /* no twisting      */
                {
                bt[0] = old_bt[0];bt[1] = old_bt[1];bt[2] = old_bt[2];
                cross_product(&b_norm[0],&bt[0],&tangent[0]);
                normalize(&b_norm[0]);
                for (j=0;j<NSET;j++)
                        {
                        alpha=  twist_angle + 6.283*j/NSET ;

        pp[total_vert][0]=ppp[i][0]+radial1*cos(alpha)*bt[0]+radial2*sin(alpha)*b_norm[0];
pp[total_vert][1]=ppp[i][1]+radial1*cos(alpha)*bt[1]+radial2*sin(alpha)*b_norm[1];
pp[total_vert][2]=ppp[i][2]+radial1*cos(alpha)*bt[2]+radial2*sin(alpha)*b_norm[2];

                        total_vert++;
                        count++;
                        }

                }/* end of no twisting       */
        else

        {       /* handle the twisting first            */
        twist = acos(twist);
        /*if (twist > (PI/2.0)  && twist <= PI ) twist -= PI/2.0;
        if (twist > PI && twist <= ( PI *3 /2.0)) twist -= PI;
        if (twist > (PI *3/2.0) && twist <= ( PI *2.0)) twist = 2*PI - twist;
twist= 0.0;*/
        /*twist_angle= twist_angle - twist;*/
        printf ("   twist_angle %f    %f \n",twist_angle,twist);
        cross_product(&bt[0],&vect01[0],&vect12[0]);
        u=dot_product(&vect00[0],&vect01[0]);
        normalize(&bt[0]);
        if (fabs(twist) > (PI/6.0)){
                bt[0] = -bt[0];
                bt[1] = -bt[1];
                bt[2] = -bt[2];
                }
        printf("%2.4f \n",u);
 bt[0] = old_bt[0];bt[1] = old_bt[1];bt[2] = old_bt[2];

        cross_product(&b_norm[0],&bt[0],&tangent[0]);

        normalize(&b_norm[0]);

        for (j=0;j<NSET;j++)
                        {
                        alpha= twist_angle  +  6.283*j/NSET ;
 pp[total_vert][0]=ppp[i][0]+radial1*cos(alpha)*bt[0]+radial2*sin(alpha)*b_norm[0];
 pp[total_vert][1]=ppp[i][1]+radial1*cos(alpha)*bt[1]+radial2*sin(alpha)*b_norm[1];
 pp[total_vert][2]=ppp[i][2]+radial1*cos(alpha)*bt[2]+radial2*sin(alpha)*b_norm[2];
                        total_vert++;
                        count++;
                        }

                }/* the end of twisting      */
        }/* end of ( i !=1 && i != p_count-1 )          */
        for (j=0;j<3;j++){
                old_bt[j]=bt[j];
                old_b_norm[j]= b_norm[j];
```

```
            }
}       /* end of i loop    */

        nvert = count;
        v_p_count[finish_yarn].v_count = count;
        printf("\n # of V %d",count);
        npoly = count-NSET;
        nzset=count/NSET;
        nvpzset=NSET;
        tmp[0]=4;tmp[1]=yarn_color;
        /***************    bottum ****************/
        k =nvpzset+ old_count;
        tmp[0]=4;

        for (i=1;i<nvpzset/2;i++)
                {
                tmp[2]=i+ old_count;
                tmp[3]=tmp[2]+1;
                tmp[4]=k-1;
                tmp[5]=k;
                k--;
                for (ii=0;ii<6;ii++)
                        index[poly_index][ii] = tmp[ii];
                poly_index++;
                v_p_count[finish_yarn].p_count++;

                }



        /***************    sids ****************/
        for (i=1;i<nzset;i++)
                {
                k=i*nvpzset+1+old_count;
                for (j=1;j<nvpzset;j++)
                        {
                        tmp[2]=j+(i-1)*nvpzset+old_count;
                        tmp[3]=k;
                        tmp[4]=k+1;
                        tmp[5]=tmp[2]+1;
                        k++;

                        for (ii=0;ii<6;ii++)
                                index[poly_index][ii]=tmp[ii];
                        poly_index++;
                        v_p_count[finish_yarn].p_count++;
                        }
                        tmp[2]=(i-1)*nvpzset+j+old_count;
                        tmp[3]=nvpzset+i*nvpzset+old_count;
                        tmp[4]=i*nvpzset+1+old_count;
                        tmp[5]=1+(i-1)*nvpzset+old_count;

                        for (ii=0;ii<6;ii++)
                                index[poly_index][ii]=tmp[ii];
                        poly_index++;
                        v_p_count[finish_yarn].p_count++;
                }
```

```
        /***************    top ***************/

        k = nvpzset*nzset+old_count;

        for (i=1;i<nvpzset/2;i++)
                {
                tmp[2] = i+nvpzset*(nzset-1) + old_count;
                tmp[3] = k;
                tmp[4] = k-1;
                tmp[5] = tmp[2]+1;
                k--;
                for (ii=0;ii<6;ii++)
                        index[poly_index][ii] = tmp[ii];
                poly_index++;
                v_p_count[finish_yarn].p_count++;
                }

         old_count + = count;


}       /* end of build yarn       */



cardsp (u,n)
        float u;
        int n;

{
        float t[4],pc[4];

 t[0]=u*u*u;
 t[1]=u*u;
 t[2]=u;
 t[3]=1.0;
 pc[0]=t[0]*pmatrix[0][0]+t[1]*pmatrix[1][0]+t[2]*pmatrix[2][0]+pmatrix[3][0];
 pc[1]=t[0]*pmatrix[0][1]+t[1]*pmatrix[1][1]+t[2]*pmatrix[2][1]+pmatrix[3][1];
 pc[2]=t[0]*pmatrix[0][2]+t[1]*pmatrix[1][2]+t[2]*pmatrix[2][2]+pmatrix[3][2];
 pc[3]=t[0]*pmatrix[0][3]+t[1]*pmatrix[1][3]+t[2]*pmatrix[2][3]+pmatrix[3][3];

          x = cp[n][0]*pc[0]+cp[n+1][0]*pc[1]+cp[n+2][0]*pc[2]+cp[n+3][0]*pc[3];
          y = cp[n][1]*pc[0]+cp[n+1][1]*pc[1]+cp[n+2][1]*pc[2]+cp[n+3][1]*pc[3];
          z = cp[n][2]*pc[0]+cp[n+1][2]*pc[1]+cp[n+2][2]*pc[2]+cp[n+3][2]*pc[3];

 } /* end of cardsp*/



 /* Cross_product,dot_product and vector normalization      */


 /* ************ Dot_product ******************/
```

```
/* Input the points of vector A and B          */
/* Return value : cos(theta)                   */

#include <math.h>

float dot_product(pva,pvb)          float *pva,*pvb;
{
float cos_theta;
        normalize(pva);
        normalize(pvb);
        cos_theta=(*(pva)**(pvb) + *(pva+1)**(pvb+1) + *(pva+2)**(pvb+2));
        return(cos_theta);

}       /* end of dot_product      */

/*************** Normalize*****************/
/* Input the point of the vector which will be normalized      */

normalize(pv)   float *pv;
{
float m;
        m= sqrt(*(pv)**(pv) + *(pv+1)**(pv+1) + *(pv+2)**(pv+2));
        *(pv) / =m;
        *(pv+1) /=m;
        *(pv+2) /=m;
} /* end of vector_normalize         */

/************ Cross_product **************/
cross_product(pvc,pva,pvb)
        float *pva,*pvb,*pvc;
{
normalize(pva);
normalize(pvb);

*pvc = *(pva+1) * *(pvb+2) - *(pva+2) * *(pvb+1);
*(pvc+1) = *(pva+2) * *pvb - *pva * *(pvb+2);
*(pvc+2) = *pva * *(pvb+1) - *(pva+1) * *pvb;
if (*pvc >= -0.001 && *pvc <= 0.001 ) *pvc = 0.0;
if (*(pvc+1) >= -0.001 && *(pvc+1) <= 0.001 ) *(pvc+1) = 0.0;
if (*(pvc+2) >= -0.001 && *(pvc+2) <= 0.001 ) *(pvc+2) = 0.0;

if ((fabs(*pvc) <=0.01 ) && (fabs(*(pvc+1))<=0.01) && (fabs(*(pvc+2))<=0.01))
        if (fabs(*(pva))<=0.01)
                {
                *pvc = -1.0;
                *(pvc+1)= 0.0;
                *(pvc+2)=0.0;
                }
        else
                {
                if (fabs(*(pva+1)) <= 0.01)
                        {
                        *pvc = 0.0;
                        *(pvc+1)= -1.0;
                        *(pvc+2)=0.0;
                        }
                }
        normalize(pvc);
```

```
} /* end of cross_product */
```

Model Depictor

```
#include <usercore.h>
#include <sun/fbio.h>
#include <math.h>
#include <stdio.h>
#include "/home/Cai/demolib.h"
#include "model.h"


int nvert,npoly;
float bbox[3][2];
float planeq[MAXPOLY][4];
float vertices [MAXVERT][3];
float n_vert[MAXVERT][3];
float normal[MAXVERT][3];
short mycolor1,cindex[MAXVERT];
int npvert[MAXPOLY];
int *pvertptr[MAXPOLY];
short plan_info[MAXPOLY];
int pvert[MAXPVERT];
float dxlist[100],dylist[100],dzlist[100];
int indxlist[100],number;
float red[256],grn[256],blu[256],dred[256],dgrn[256],dblu[256];
int num_shade_color,num_shade_level;
float forward;

int vertex_index[6][4]={{1,2,6,5},
                        {2,3,7,6},
                        {3,4,8,7},
                        {4,1,5,8},
                        {1,4,3,2},
                        {5,6,7,8}};

/* line variables */
int nline,nlvert,line_index_list[MAXLVERT],npline[MAXLINE];
short cline[MAXLINE];
float line_vert[MAXLVERT][3];

float xmax,xmin,ymax,ymin,zmax,zmin,xcent,ycent,zcent,length,emin,emax,scale;
float xlist[100],ylist[100],zlist[100];


float menu_x,menu_y,menu_h,menu_w,menu_f,window_fact;
int     n_button,face_remove;
float T,B,L,R,dot,alpha,beta,r;

int     nobj,npoly,nvert;
int     free_vert,free_poly,free_obj,free_pvert;
int     old_vert,old_poly,old_pvert;
int raster_id;

struct  menu_table_format menu_table[30];

struct  view_parameters_format          my_view_parameters;
struct  image_status_format image_status;
```

```
struct obj *objlst;
struct poly *polylst;
struct vert *vertlst;
struct pvert *pvertlst;
VECTOR cut_normal;
POINT plane_locate;
POINT cut_plane[6];

main(argc, argv)
int argc;
char *argv[];
{
char string[81];
int i,end,j,k;
int button,length =0;
float gema,w,cosb,x,y,z;
POINT cut_point;
LINE line;
FILE *fptr;

printf("Input cut plane normal ");
scanf("%f %f %f",&cut_normal.u,&cut_normal.v,&cut_normal.w);

if ( load_file(argv[1]) != NULL ) exit(1);



get_view_surface(our_surface,argv);
start_up_core();
init_menu();

/* initialize */

r=5000.0;
alpha=beta=0.0;
face_remove=FALSE;
window_fact=1.0;
num_shade_color=4,num_shade_level=52;
image_status.line=image_status.wire_frame=
image_status.out_box=image_status.solid=
image_status.special=image_status.cutting =image_status.move = FALSE;
forward = 12.0;
raster_id = 900;
plane_locate.x = 0.0;plane_locate.y = 0.0;plane_locate.z = 0.0;
/*
image_status.out_box=TRUE;

create_retained_segment(OUTBOX);
draw_box();
close_retained_segment(OUTBOX);
*/
new_view_point(alpha,beta,r);
for (;;)
{
switch(call_menu(MENU,TRUE))
        {
```

```
case 1: /* Right + 10        */
          if (image_check()) break;
          alpha +=10.0;
           new_view_point(alpha,beta,r);
           image_switch();
break;

case 20: /* Right + 2        */
          if (image_check()) break;
          alpha +=2.0;
          new_view_point(alpha,beta,r);
          image_switch();
break;

case 2: /* Left -10          */
if (image_check()) break;

  alpha - =10.0;
  new_view_point(alpha,beta,r);
  image_switch();

  break;

 case 21: /* Left -2         */
if (image_check()) break;
  alpha - =2.0;
  new_view_point(alpha,beta,r);
  image_switch();
  break;


 case 3: /*          down +10        */
if (image_check()) break;
  if (beta <90.0) beta +=10.0;
  new_view_point(alpha,beta,r);
  image_switch();
 break;

 case 23: /*         down +2         */
 if (image_check()) break;

   if (beta <90.0) beta +=2.0;
   new_view_point(alpha,beta,r);
 image_switch();


  break;

  case 4: /*          up +10  */
  if (image_check()) break;

    if (beta > -90.0 ) beta -=10.0;
    new_view_point(alpha,beta,r);
           image_switch();

  break;
```

```
case 22: /*          up +2     */
if (image_check()) break;

 if (beta > -90.0 ) beta -=2.0;
  new_view_point(alpha,beta,r);
  image_switch();

break;

case 5:
if (image_check()) break;

 r +=2500.0;        /* zoom out        */

  new_view_point(alpha,beta,r);
  image_switch();

break;

case 6:
if (image_check()) break;

 r -=2500.0;        /* zoom in          */
 if (r<100.0) r=100.0;
  new_view_point(alpha,beta,r);
  image_switch();

break;

case 7: /* reset     */

if (image_check()) break;
 r =5000.0;
 alpha=beta=0.0;
  window_fact=1.0;
  setvwpo(x,y,z,bbox);
  new_view_point(alpha,beta,r);
  image_switch();

break;

case 8:  /************* solid **********************/

          if (face_remove || image_status.move) {
                    new_frame();
                    face_remove =  image_status.move = FALSE;
                    }

          new_view_point(alpha,beta,r);
          set_shading_parameters(.01,.9,.0,0.0,7.,0,1);
          set_primitive_attributes(&PRIMATTS);
          set_polygon_interior_style( SHADED );
          create_temporary_segment();
          drawobj1();
          face_remove=TRUE;
          close_temporary_segment();

          break;
```

```
case 9:  /******************** cut **********************/
        if (face_remove ){
                new_frame();
                face_remove = FALSE;
                }

        if (image_status.wire_frame){
                image_status.wire_frame=FALSE;
                delete_retained_segment(WIREFRAME);
                }
        if (image_status.line){
                image_status.line=FALSE;
                delete_retained_segment(DRAWLINE);
                }
        if (image_status.out_box){
                image_status.out_box=FALSE;
                delete_retained_segment(OUTBOX);
                }

        if (image_status.move || image_status.cutting){
                if (image_status.move) {
                        delete_retained_segment(SCREENTEXT);
                        image_status.move = FALSE;
                        }
                if (image_status.cutting)image_status.cutting = FALSE;
                new_frame();
                delete_retained_segment(PLANE);
                }


        cut_obj(cut_normal,plane_locate);
        create_temporary_segment();
        draw_box();
        /*draw_wireframe();*/
        close_temporary_segment();
        create_retained_segment(PLANE);
        new_cut_plane(cut_normal,plane_locate,&cut_plane[0].x,&i);
        draw_cut_plane(i);
        close_retained_segment(PLANE);

        rebuild_planeq();
        image_status.cutting = TRUE;

        section_draw();
        /*inquire_viewing_parameters(&my_view_parameters);
 set_view_reference_point(0.0,0.0,0.0);*/



        break;



case 10:
        shut_down_core1();
        exit();
case 11:
```

```
        if (image_status.wire_frame)
                {
                image_status.wire_frame=FALSE;
                delete_retained_segment(WIREFRAME);
                }
        else
                {
                image_status.wire_frame=TRUE;
                new_view_point(alpha,beta,r);
                create_retained_segment(WIREFRAME);

        draw_wireframe();
                 close_retained_segment(WIREFRAME);
                }
        break;

case 12:

                if (image_status.out_box)
                {
                image_status.out_box=FALSE;
                delete_retained_segment(OUTBOX);
                }
         else
                {
                image_status.out_box=TRUE;
                new_view_point(alpha,beta,r);
                create_retained_segment(OUTBOX);
         draw_box();
                 close_retained_segment(OUTBOX);
                 }
         break;

 case 13:

                 if (image_status.line)      /* line   */
                 {
                 image_status.line=FALSE;
                 delete_retained_segment(DRAWLINE);
                 }
          else
                 {
                 image_status.line=TRUE;
                  new_view_point(alpha,beta,r);
                 create_retained_segment(DRAWLINE);
                 drawline();
                  close_retained_segment(DRAWLINE);
                 }
          break;
 case 14:          /* larger*2        */
 if (window_fact>0.1 ) window_fact -=0.1; else if (window_fact>0.01) window_fact -=0.01;
         new_view_point(alpha,beta,r);
   image_switch();
         break;

 case 24:          /* larger*1        */
         if (window_fact>0.01) window_fact -=0.01;
```

```
        new_view_point(alpha,beta,r);
 image_switch();
        break;

case 15:        /* smaller*2       */
        window_fact +=0.1;
        new_view_point(alpha,beta,r);
 image_switch();

        break;

case 25:        /* smaller*1       */
        window_fact +=0.01;
        new_view_point(alpha,beta,r);
 image_switch();
        break;


case 18:                /* Switch cutting direction          */

                cut_normal.u = -cut_normal.u;
                cut_normal.v = -cut_normal.v;
                cut_normal.w = -cut_normal.w;

break;
case 19:        /* Cutting plane move forward     */
        if (bbox[0][1]<=plane_locate.x &&
                bbox[1][1]<=plane_locate.y &&
                bbox[2][1]<=plane_locate.z ) break;
        move_cut_plane(1);
        break;
 case 26:       /* Cutting plane move backward   */
        if (bbox[0][0]>=plane_locate.x &&
                bbox[1][0]>=plane_locate.y &&
                bbox[2][0]>=plane_locate.z ) break;
        move_cut_plane(-1);
        break;
 case 27:       /* Raster file      */
        raster_file();
        break;
 case 28:       /* Load Raster file */
        load_raster();
        break;
 case 29:       /* Move Raster  */
        move_raster();
        break;
 case 30:       /* Delete Raster  */


 re_store();
        delete_raster();
        break;
 case 31:       /* Menu hiding  */
        set_segment_visibility(MENU,FALSE);
        set_echo(LOCATOR, 1 ,1);
        for (;;)
        {
        do
```

```
                await_any_button_get_locator_2(20000000,1,&button,&x,&y);
        while(button == 0);
        if(button==1 || button ==2 || button == 3) break;

        }
        set_segment_visibility(MENU,TRUE);
        set_echo(LOCATOR,1,0);

        break;
case 32:        /* cutting file */

        if (face_remove ){
                new_frame();
                face_remove = FALSE;
                }

        if (image_status.wire_frame){
                image_status.wire_frame=FALSE;
                delete_retained_segment(WIREFRAME);
                }
        if (image_status.line){
                image_status.line=FALSE;
                delete_retained_segment(DRAWLINE);
                }
        if (image_status.out_box){
                image_status.out_box=FALSE;
                delete_retained_segment(OUTBOX);
                }

        if (image_status.move || image_status.cutting){
                if (image_status.move) image_status.move = FALSE;
                if (image_status.cutting)image_status.cutting = FALSE;
                new_frame();
                delete_retained_segment(PLANE);
                }
 if ((fptr = fopen("cut_file", "r")) == NULL) {
                printf("Can't open cut_file file\n");
                exit (1);
                }


  fscanf(fptr,"%d",&k);
          create_retained_segment(PLANE);
          draw_box();
          for(j=0;j<k;j++){
          fscanf(fptr,"%f%f%f",&cut_normal.u,&cut_normal.v,&cut_normal.w);
          fscanf(fptr,"%f%f%f",&plane_locate.x,&plane_locate.y,&plane_locate.z);
          plane_locate.x =plane_locate.x*scale;
          plane_locate.y =plane_locate.y*scale;
          plane_locate.z =plane_locate.z*scale;
          cut_obj(cut_normal,plane_locate);
          new_cut_plane(cut_normal,plane_locate,&cut_plane[0].x,&i);
          draw_cut_plane(i);
          rebuild_planeq();
          }

  close(fptr);
```

```
        close_retained_segment(PLANE);
        image_status.cutting = TRUE;

        /*section_draw();*/



        break;/* end of cut files */
default:
break;

        }       /* end of switch  */

}       /* end of waiting for      */

}       /* end of main    */

int image_check()
{
if (image_status.move || image_status.cutting) {
        delete_retained_segment(PLANE);
        new_frame();
        image_status.cutting = image_status.move = FALSE;
        }

if (image_status.cutting) {
        new_frame();
        image_status.cutting = FALSE;
        }
if (image_status.line== FALSE && image_status.wire_frame
        == FALSE && image_status.out_box== FALSE && image_status.solid
        == FALSE && image_status.cutting == FALSE) return(TRUE);
else
                return(FALSE);
}

new_view_point(alpha,beta,r)
float alpha,beta,r;
{
float cosb,x,y,z;

cosb= fabs(cos(beta*PI/180.0));
x = r*cosb*sin(alpha*PI/180.0);
y = r*sin(beta*PI/180.0);
z = r*cosb*cos(alpha*PI/180.0);
setvwpo(x,y,z,bbox);
/* set_single_window;    */
}       /* end of new_view_point          */


move_cut_plane(direction) int direction;
{int i;
char buff[120];


        if (image_status.cutting){
                delete_retained_segment(PLANE);
```

```
            image_status.cutting = FALSE;
            }
        if (image_status.move== FALSE &&
            face_remove == FALSE && image_status.wire_frame == FALSE ){
            create_temporary_segment();
            draw_box();
            /*draw_wireframe();*/

            close_temporary_segment();
            create_retained_segment(PLANE);
            }
        if (image_status.move== FALSE &&
            (face_remove  ||image_status.wire_frame  )){
            create_temporary_segment();
            draw_box();
            close_temporary_segment();
            create_retained_segment(PLANE);
            }
        plane_locate.x += forward*cut_normal.u * direction ;
        plane_locate.y += forward*cut_normal.v * direction ;
        plane_locate.z += forward*cut_normal.w * direction ;
        new_cut_plane(cut_normal,plane_locate,&cut_plane[0].x,&i);
        if (image_status.move  ){
            delete_retained_segment(PLANE);
            create_retained_segment(PLANE);
            delete_retained_segment(SCREENTEXT);
            }
        draw_cut_plane(i);
        close_retained_segment(PLANE);

        inquire_viewing_parameters(&my_view_parameters);
        set_view_reference_point(0.0,0.0,0.0);
        set_view_plane_normal(0.0,0.0, -1.0);
        set_view_plane_distance(0.0);
        set_projection(PARALLEL,0.0,0.0,1.0);
        set_view_up_3(0.0,1.0,0.0);
        set_window(0.0,100.0,0.0,767.0);
        set_view_depth(0.0,1.0);
        set_window_clipping(FALSE);
        set_viewport_3(0.0,0.915,0.0,.75,0.0,1.0);

    create_retained_segment(SCREENTEXT);
    move_abs_3(10.0, 10.0,0.0);
    text("Cutting Plane Location: ");

        sprintf(buff,"%3.3f  %3.3f  %3.3f",
                plane_locate.x/scale,plane_locate.y/scale,plane_locate.z/scale);
        move_abs_3(35.0, 10.0,0.0);
        text(buff);
        close_retained_segment(SCREENTEXT);

        set_viewing_parameters(&my_view_parameters);


        image_status.move= TRUE;
}/* end of move_cut_plane */
```

```
#include "mycut"

raster_file()
{
int rasfid,i,j;
int butnum,button;
float xmin, ymin, x, y,z, xmax, ymax,zmax,mx,my;
float wx, wy, wz,wwx, wwy,wwz,kx,ky,xmin_ndc,xmax_ndc,ymin_ndc,ymax_ndc;
 char string[80]; int length;
struct suncore_raster my_raster;
extern struct  vwsurf *our_surface;
struct { int type, nbytes;
         char *data; } map;

FILE *fptr;
char *path_name;

         path_name = "/home/Cai/load_file_lib";
 inquire_viewing_parameters(&my_view_parameters);
 set_view_reference_point(0.0,0.0,0.0);

         set_view_plane_normal(0.0,0.0, -1.0);
         set_view_plane_distance(0.0);
         set_projection(PARALLEL,0.0,0.0,1.0);
         set_view_up_3(0.0,1.0,0.0);
         set_window(0.0,100.0,0.0,767.0);
         set_view_depth(0.0,1.0);
         set_window_clipping(FALSE);
         set_viewport_3(0.0,0.915,0.0,.75,0.0,1.0);



   set_echo(LOCATOR, 1 ,1);

   for (;;)
         {
         do
                  await_any_button_get_locator_2(1000000,1,&button,&mx,&my);
         while(button == 0);

                  break;

         }
 set_echo_position( LOCATOR, 1, mx,my);

 printf("\nfirst point  %f  %f",mx,my);
 map_ndc_to_world_3( x, y, 0.0,&wx, &wy,&wz);
 printf("xyz %f %f %f",wx, wy, wz);
 set_echo( LOCATOR,1, 6);
   do {
          await_any_button_get_locator_2( 1000000,1, &butnum, &xmax, &ymax);

   } while (butnum != 3);
   printf("\nfirst point  %f  %f",xmax, ymax);
   printf("\n%f %f %f",bbox[0][0],bbox[1][0],bbox[2][0]);
   printf("\n%f %f %f",bbox[0][1],bbox[1][1],bbox[2][1]);
  set_echo_position( KEYBOARD,1, 0.25, 0.1);        /* move to start positn */
```

```
  set_keyboard( 1, 80, "", 0);                    /* set user char buf size */
  xmin_ndc = (mx<=xmax ) ? mx : xmax;
  xmax_ndc = (xmax>= mx) ? xmax : mx;
  ymin_ndc = (my<=ymax ) ? my : ymax;
  ymax_ndc = (ymax>= my) ? ymax : my;
   size_raster(our_surface,xmin_ndc,xmax_ndc,ymin_ndc,ymax_ndc,&my_raster);
        allocate_raster(&my_raster);
        get_raster(our_surface,xmin_ndc,xmax_ndc,ymin_ndc,ymax_ndc,0,0,&my_raster);

  create_retained_segment(SCREENTEXT);

  move_abs_2(10.0, 10.0,0.0);
  text("Raster File Name: ");
  set_echo( KEYBOARD,1,1);                              /* echo the text */
  await_keyboard( 10000000,1, string, &length);

  if (length && string[length-1] == '\n') string[length-1] = '\0';

move_abs_2(25.0, 10.0,0.0);
text( string);
move_abs_2(50.0,50.0);

close_retained_segment(SCREENTEXT);

set_image_transformation_type( XLATE2);
create_retained_segment(99);
set_primitive_attributes( &PRIMATTS);
set_pick_id( 1);
put_raster(&my_raster);
set_segment_detectability( 99, 5);
if( (rasfid = open( "try", 0)) == -1) {
            rasfid = creat( "try", 0755);
           }
         if (rasfid != -1) {
            map.type = 1;  map.nbytes = 768;  map.data = "85";
            file_to_raster( my_raster, &map, rasfid );
            close( rasfid);
           }
         free_raster(&my_raster);

close_retained_segment(99);
strcat(path_name,&string[0]);
save_segment(99,string);

delete_retained_segment(99);
delete_retained_segment(SCREENTEXT);

set_viewing_parameters(&my_view_parameters);

}/* end of raster_file */

load_raster()
{
int rasfid,i,j;
int butnum,button;
float xmin, ymin, x, y,z, xmax, ymax,zmax,mx,my;
float wx, wy, wz,wwx, wwy,wwz,kx,ky,xmin_ndc,xmax_ndc,ymin_ndc,ymax_ndc;
 char string[80]; int length;
```

```
struct suncore_raster my_raster;
extern struct  vwsurf *our_surface;
struct { int type, nbytes;
        char *data; } map;
FILE *fptr;

int segnam, pickid,  segtype;
   float sx0, sy0, ang0, tx0, ty0;
   float sx, sy, ang, tx, ty;
   float  px, py;
         char *path_name;


         inquire_viewing_parameters(&my_view_parameters);
         set_view_reference_point(0.0,0.0,0.0);

         set_view_plane_normal(0.0,0.0, -1.0);
         set_view_plane_distance(0.0);
         set_projection(PARALLEL,0.0,0.0,1.0);
         set_view_up_3(0.0,1.0,0.0);
         set_window(0.0,100.0,0.0,767.0);
         set_view_depth(0.0,1.0);
         set_window_clipping(FALSE);
         set_viewport_3(0.0,0.915,0.0,.75,0.0,1.0);

         set_echo_position( KEYBOARD,1, 0.25, 0.1);
    set_keyboard( 1, 80, "", 0);
    create_retained_segment(SCREENTEXT);
    move_abs_3(10.0, 10.0,0.0);
    text("Raster File Name: ");
          await_keyboard( 10000000,1, string, &length);
    set_pick_id( 1);
    if (length && string[length-1] == '\n') string[length-1] = '\0';

          move_abs_3(25.0, 10.0,0.0);
          text( string);
          close_retained_segment(SCREENTEXT);

          if ((fptr = fopen(string, "r")) == NULL) {
                    move_abs_3(25.0, 20.0,0.0);

                    text( "Can't open raster file ");
                    for (i=0;i<10000;i++);
                    delete_retained_segment(SCREENTEXT);

                                        return(1);
                    }


          restore_segment(raster_id,string);


     set_echo(LOCATOR, 1 ,1);

     for (;;)
          {
          do
                    await_any_button_get_locator_2(1000000,1,&button,&mx,&my);
```

```
        while(button == 0);

                break;

        }
  set_echo( LOCATOR,1, 0);
  do {
        await_any_button_get_locator_2( 0,1, &butnum, &x, &y);
                tx = x-mx;  ty = y-my;

           set_segment_image_translate_2(raster_id , tx,ty);
        } while (butnum != 3);
raster_id++;
set_viewing_parameters(&my_view_parameters);
delete_retained_segment(SCREENTEXT);

}/* end of load raster */

move_raster()
{
int rasfid,i,j;
int butnum,button;
float xmin, ymin, x, y,z, xmax, ymax,zmax,mx,my;
float wx, wy, wz,wwx, wwy,wwz,kx,ky,xmin_ndc,xmax_ndc,ymin_ndc,ymax_ndc;
 char string[80]; int length;

int segnam, pickid,  segtype;
   float sx0, sy0, ang0, tx0, ty0;
   float sx, sy, ang, tx, ty;
   float  px, py;

/* inquire_viewing_parameters(&my_view_parameters);
 set_view_reference_point(0.0,0.0,0.0);


          set_view_plane_normal(0.0,0.0, -1.0);
          set_view_plane_distance(0.0);
          set_projection(PARALLEL,0.0,0.0,1.0);
          set_view_up_3(0.0,1.0,0.0);
          set_window(0.0,100.0,0.0,767.0);
          set_view_depth(0.0,1.0);
          set_window_clipping(FALSE);
          set_viewport_3(0.0,0.915,0.0,.75,0.0,1.0);

    create_retained_segment(SCREENTEXT);
    move_abs_2(10.0, 10.0);
    text("Find your image and move");
          close_retained_segment(SCREENTEXT);

          await_pick( 1000000000, 1, &segnam, &pickid);
          await_any_button_get_locator_2( 0,1, &butnum, &px, &py);

    set_echo( LOCATOR,1, 0);
    do {
          await_any_button_get_locator_2( 0,1, &butnum, &x, &y);
                tx = x-px;  ty = y-py;

            set_segment_image_translate_2(segnam , tx,ty);
```

```
        } while (butnum != 3);

set_viewing_parameters(&my_view_parameters);
delete_retained_segment(SCREENTEXT);
*/

}

delete_raster()
{
new_frame();
}



section_draw()
{
int i,j,k;
int index,flag;
float vect0[3],vect1[3],scale,plane[8][3],area,cut_area,point[400][3],u,v,w;
FILE *fptr;
float x,y,z,xc,yc,zc,sina,cosa,sinb,cosb,minx,maxx,minz,maxz,fact_x,fact_z,fact;

if ((fptr = fopen("nc.dat", "r")) == NULL) {
                        printf("Can't open nc.dat file\n");
                        exit (1);
                        }

 fscanf(fptr,"%f%f%f",&vect0[0],&vect0[1],&vect0[2]);
 fscanf(fptr,"%f",&scale);
 fscanf(fptr,"%d",&k);
 u=vect0[0];
 v=vect0[1];
 w=vect0[2];
 sinb= (u==0.0 && v==0.0) ?  0.0 : -u/sqrt(u*u+v*v) ;
 cosb= (u==0.0 && v==0.0) ? 1.0 : v/sqrt(u*u+v*v);

 cosa= (u==0.0 && v==0.0 && w==0.0) ?  1.0 : sqrt(u*u+v*v)/sqrt(u*u+v*v+w*w);
 sina= (u==0.0 && v==0.0 && w==0.0) ? 0.0 : w/sqrt(u*u+v*v+w*w);
 maxx=minx=maxz=minz=0.0;
 for (i=0;i<k;i++)
          {
                        fscanf(fptr,"%f%f%f",&plane[i][0],&plane[i][1],&plane[i][2]);
                        plane[i][0] = plane[i][0]/scale;
                        plane[i][1] = plane[i][1]/scale;
                        plane[i][2] = plane[i][2]/scale;

          }



  for (i=0;i<k;i++)
          {

                        if (i==0){
                                xc=plane[i][0];
                                yc=plane[i][1];
                                zc=plane[i][2];
```

```
                }
                xc=yc=zc=0.0;
                plane[i][0]=plane[i][0]-xc;
                plane[i][1]=plane[i][1]-yc;
                plane[i][2]=plane[i][2]-zc;
                x=plane[i][0]*cosb+plane[i][1]*sinb;
                y= -plane[i][0]*sinb+plane[i][1]*cosb;
                z= plane[i][2];
                plane[i][0] = x;plane[i][1] = y;plane[i][2] = z;
                x=plane[i][0];
                y=plane[i][1]*cosa+plane[i][2]*sina;
                z= -plane[i][1]*sina+plane[i][2]*cosa;

                plane[i][0] = x;plane[i][1] = y;plane[i][2] = z;
                plane[i][0]=plane[i][0]+xc;
                plane[i][1]=plane[i][1]+yc;
                plane[i][2]=plane[i][2]+zc;
                printf(" %3.4f %3.4f %3.4f\n",plane[i][0],plane[i][1],plane[i][2]);
                if(i==0){
                        maxx=minx=plane[i][0];
                        maxz=minz=plane[i][2];
                        }
                else {
                        maxx= (maxx<plane[i][0]) ? plane[i][0] : maxx;
                        minx= (minx>plane[i][0]) ? plane[i][0] : minx;
                        maxz= (maxz<plane[i][2]) ? plane[i][2] : maxz;
                        minz= (minz>plane[i][2]) ? plane[i][2] : minz;
                        }

        }

flag = (fabs(maxx-minx)>=fabs(maxz-minz)) ? TRUE : FALSE;

fact= (flag) ? fabs(5.4/(maxx-minx)) : fabs(5.4/(maxz-minz));
inquire_viewing_parameters(&my_view_parameters);
 set_view_reference_point(0.0,0.0,0.0);

        set_view_plane_normal(0.0,0.0, -1.0);
        set_view_plane_distance(0.0);
        set_projection(PARALLEL,0.0,0.0,1.0);
        set_view_up_3(0.0,1.0,0.0);
        set_window(0.0,6.0,0.0,6.0);
        set_view_depth(0.0,1.0);
        set_window_clipping(TRUE);

        /*set_viewport_3(0.0,0.375,0.375,0.75,0.0,1.0);*/
        set_viewport_3(0.0,0.5,0.25,0.75,0.0,1.0);
 create_temporary_segment();
 /*xlist[0]=0.0;zlist[0]=0.0;
 xlist[1]=6.0;zlist[1]=0.0;
 xlist[2]=6.0;zlist[2]=6.0;
 xlist[3]=0.0;zlist[3]=6.0;
 xlist[4]=0.0;zlist[4]=0.0;
 move_abs_2(xlist[0],zlist[0]);
 polyline_abs_2(xlist,zlist,5);*/

        set_linewidth(0.5);
        set_line_index(MENU_TEXT_COLOR);
```

```
        for (j=0;j<kj++)
                        {
                        if (flag){
                                xlist[j] = (plane[j][0]-minx)*fact+(6.0-(maxx-minx)*fact)*0.5;
                                zlist[j] = (plane[j][2]-minz)*fact+(6.0-(maxz-minz)*fact)*0.5;
                                }
                        else {
                                zlist[j] = (plane[j][0]-minx)*fact+(6.0-(maxx-minx)*fact)*0.5;
                                xlist[j] = (plane[j][2]-minz)*fact+(6.0-(maxz-minz)*fact)*0.5;
                        }
                }
        xlist[j] = xlist[0];
        zlist[j] = zlist[0];
        move_abs_2(xlist[0],zlist[0]);
        polyline_abs_2(xlist,zlist,k+1);



cut_area=0.0;



while ( fscanf(fptr,"%d",&k)!=EOF) {

        fscanf(fptr,"%f%f%f",&vect0[0],&vect0[1],&vect0[2]);
        fscanf(fptr,"%d",&k);

        for (i=0;i<k;i++)
        {
                fscanf(fptr,"%f%f%f",&point[i][0],&point[i][1],&point[i][2]);
                point[i][0] = point[i][0]/scale;
                point[i][1] = point[i][1]/scale;
                point[i][2] = point[i][2]/scale;
                xc=yc=zc=0.0;
                point[i][0]=point[i][0]-xc;
                point[i][1]=point[i][1]-yc;
                point[i][2]=point[i][2]-zc;
                x=point[i][0]*cosb+point[i][1]*sinb;
                y= -point[i][0]*sinb+point[i][1]*cosb;
                z= point[i][2];
                point[i][0] = x;point[i][1] = y;point[i][2] = z;
                x=point[i][0];
                y=point[i][1]*cosa+point[i][2]*sina;
                z= -point[i][1]*sina+point[i][2]*cosa;

                point[i][0] = x;point[i][1] = y;point[i][2] = z;
                point[i][0]=point[i][0]+xc;
                point[i][1]=point[i][1]+yc;
                point[i][2]=point[i][2]+zc;
                /*printf(" %d %3.4f %3.4f %3.4f\n",i,point[i][0],point[i][1],point[i][2]);*/

                if (flag){
                                xlist[i] = (point[i][0]-minx)*fact+(6.0-(maxx-minx)*fact)*0.5;
                                zlist[i] = (point[i][2]-minz)*fact+(6.0-(maxz-minz)*fact)*0.5;
                                }
                        else {
                                zlist[i] = (point[i][0]-minx)*fact+(6.0-(maxx-minx)*fact)*0.5;
```

```
                                xlist[i] = (point[i][2]-minz)*fact+(6.0-(maxz-minz)*fact)*0.5;

                        }

        }
        xlist[i] = xlist[0];
        zlist[i] = zlist[0];
        move_abs_2(xlist[0],zlist[0]);
        polyline_abs_2(xlist,zlist,k+1);


}/* end of cut file */
set_linewidth(0.0);
close_temporary_segment();
set_viewing_parameters(&my_view_parameters);

close (fptr );
}/* end of section draw */



re_store()
{
int i,j,k,index,num_vert,poly_index,new_vert,new_index,new_nobj;
float x,y,z,x0,y0,z0,length,bbox1[3][2];
short *ptr,*vert_index_lst;
struct vert *vertlstptr;
FILE *fptr;


        new_nobj=new_vert=0;
        vert_index_lst = (short *)malloc(free_vert * sizeof(short));
        for (i=0;i<free_vert;i++)
                vert_index_lst[i]= -1;
        fptr = fopen("nn.dat", "w");

        for (i=0;i<nobj;i++) if (objlst[i].npoly !=0) new_nobj++;
        npoly=0;
        for (i=0;i<nobj;i++) npoly=npoly +objlst[i].npoly;

        for (i=0;i<nobj;i++){
                poly_index = objlst[i].index;
                for (j=0;j<objlst[i].npoly;j++){
                        linking(&poly_index);
                        ptr = polylst[poly_index].pvert.ptr;
                        num_vert = polylst[poly_index++].npvert;
                        for (k=0;k<num_vert;k++){
                                                vert_index_lst[*ptr++]=-2;
                                        }/* mark all the vertexes of one polygon */
                        }/* end of j poly loop      */
        }/* end of i loop */
        for (k=0;k<free_vert;k++)
                                                if (vert_index_lst[k]==-2) new_vert++;

        bbox1[0][0] = bbox1[0][1] = bbox1[1][0] =
        bbox1[1][1] =  bbox1[2][0] = bbox1[2][1] = 0.0;
```

```
for (i=0;i<free_vert;i++){
        if(vert_index_lst[i]== -2){
                if (bbox1[0][0] >= vertlst[i].vertex.x) bbox1[0][0]
                        = vertlst[i].vertex.x;
                if (bbox1[0][1] < vertlst[i].vertex.x) bbox1[0][1]
                        = vertlst[i].vertex.x;
                if (bbox1[1][0] >= vertlst[i].vertex.y) bbox1[1][0]
                        = vertlst[i].vertex.y;
                if (bbox1[1][1] < vertlst[i].vertex.y) bbox1[1][1]
                        = vertlst[i].vertex.y;
                if (bbox1[2][0] >= vertlst[i].vertex.z) bbox1[2][0]
                        = vertlst[i].vertex.z;
                if (bbox1[2][1] < vertlst[i].vertex.z) bbox1[2][1]
                        = vertlst[i].vertex.z;
                        }
        vert_index_lst[i]= -1;

        }
fprintf(fptr,"%d %d %d\n",new_nobj,npoly,new_vert);

for (i=0;i<3;i++) for(j=0;j<2;j++) bbox1[i][j] * = 1.2;

fprintf(fptr, "%f %f %f %f %f %f\n", bbox1[0][0]/scale, bbox1[0][1]/scale,
 bbox1[1][0]/scale,bbox1[1][1]/scale,bbox1[2][0]/scale, bbox1[2][1]/scale);

new_vert=0;
for (i=0;i<nobj;i++){
        if (objlst[i].npoly ==0) goto next_obj;
        poly_index = objlst[i].index;
        for (k=0;k<free_vert;k++)
                        vert_index_lst[k]= -1;

        for (j=0;j<objlst[i].npoly;j++){
                linking(&poly_index);
                ptr = polylst[poly_index].pvert.ptr;
                num_vert = polylst[poly_index++].npvert;
                for (k=0;k<num_vert;k++){
                                        vert_index_lst[*ptr++]=-2;
                                }/* mark all the vertexes of one polygon */
                }/* end of j poly loop      */

        num_vert=0;
        for (k=0;k<free_vert;k++)
                                        if (vert_index_lst[k]==-2) num_vert++;

        fprintf(fptr,"%d %d\n",objlst[i].npoly,num_vert);
        for (k=0;k<free_vert;k++){
                                        if (vert_index_lst[k]==-2){
                                        x=vertlst[k].vertex.x/scale;
                                        y=vertlst[k].vertex.y/scale;
                                        z=vertlst[k].vertex.z/scale;
                                        fprintf(fptr, "%f %f %f\n",x,y,z);
                                        vert_index_lst[k]=new_vert;
                                        new_vert++;
                                        }
                }/* end of sending          */

        poly_index = objlst[i].index;
```

```
                for (j=0;j<objlst[i].npoly;j++){
                        linking(&poly_index);
                        ptr = polylst[poly_index].pvert.ptr;
                        num_vert = polylst[poly_index].npvert;
                        fprintf(fptr,"%d ",num_vert);
                        fprintf(fptr,"%d ",polylst[poly_index++].info);
                        for (k=0;k<num_vert;k++){
                                                new_index=vert_index_lst[*ptr++]+1;
                                                fprintf(fptr,"%d ",new_index);

                                }/* send all the vertexes index of one polygon */
                        fprintf(fptr,"\n");
                        }/* end of j poly loop      */
        next_obj:k=0; /* no poly !! */
        }/* end of i obj loop       */
        free(vert_index_lst);
        fprintf(fptr,"%d %d\n",0,0);
        fclose(fptr);
} /*     end of restore    */
```

Subroutines for Model Depiction

```
#include <sun/fbio.h>
#include <usercore.h>
#include <math.h>
#include <stdio.h>
#include "model.h"


/* Externals */
extern  nobj,npoly,nvert,vertex_index[6][4];
extern  free_vert,free_poly,free_obj,free_pvert;
extern  old_vert,old_poly,old_pvert;
extern  float bbox[3][2];

/* Externals */

extern struct obj *objlst;
extern struct poly *polylst;
extern struct vert *vertlst;
extern struct pvert *pvertlst;


int plane_cut_line(normal,locate,line,cut_point)
VECTOR normal;
POINT locate;
LINE line;
POINT *cut_point;
{
float px,py,pz,d,d1,d2,cosa0,cosa1,delta;
float a[3][3];
VECTOR v0,v1,l,vtemp;
int end0_in,end1_in;

float  dot_product();

v0.u = line.p0.x - locate.x;
v0.v = line.p0.y - locate.y;
```

```
v0.w = line.p0.z - locate.z;
v1.u = line.p1.x - locate.x;
v1.v = line.p1.y - locate.y;
v1.w = line.p1.z - locate.z;
cosa0 = dot_product(&normal,&v0);
cosa1 = dot_product(&normal,&v1);

if (cosa0> PLANETHICK && cosa1>PLANETHICK ) return(NOCUT);
end0_in = (cosa0 <= PLANETHICK &&  cosa0 >= -PLANETHICK) ? TRUE :FALSE;
end1_in = (cosa1 <= PLANETHICK &&  cosa1 >= -PLANETHICK) ? TRUE :FALSE;

if (end1_in &&
        end0_in) return(ONPLANE);
if ((cosa0< -PLANETHICK && cosa1< -PLANETHICK )
        | |( end0_in && cosa1<  -PLANETHICK)
        | | ( end1_in && cosa0< -PLANETHICK ) )
         return(ALLCUT);

if ((end0_in && cosa1>PLANETHICK)
        | | (end1_in && cosa0>PLANETHICK)){
        if (end0_in){      /* end at p0       */
                cut_point->x = line.p0.x;
                cut_point->y = line.p0.y;
                cut_point->z = line.p0.z;
                return(ENDCUT0);
                }
        if (end1_in){      /* end at p1       */
                cut_point->x = line.p1.x;
                cut_point->y = line.p1.y;
                cut_point->z = line.p1.z;
                return(ENDCUT1);
                }

        }


 l.u = line.p1.x - line.p0.x;
 l.v = line.p1.y - line.p0.y;
 l.w = line.p1.z - line.p0.z;
 delta = ((line.p1.x + 10.0 != line.p0.x) && (line.p1.x + 10.0 != line.p0.x)
          && (line.p1.x + 10.0 != line.p0.x))  ? 10.0 : 20.0;
 vtemp.u = line.p1.x + delta  - line.p0.x;
 vtemp.v = line.p1.y + delta  - line.p0.y;
 vtemp.w = line.p1.z + delta  - line.p0.z;


 cross_product(&v0,&vtemp,&l);
 cross_product(&v1,&v0,&l);

 a[0][0] = normal.u; a[0][1] = normal.v; a[0][2] = normal.w;
 a[1][0] = v0.u; a[1][1] = v0.v; a[1][2] = v0.w;
 a[2][0] = v1.u; a[2][1] = v1.v; a[2][2] = v1.w;

 d = normal.u*locate.x+normal.v*locate.y+normal.w*locate.z;
 d1 = v0.u*line.p1.x+v0.v*line.p1.y+v0.w*line.p1.z;
 d2 = v1.u*line.p1.x+v1.v*line.p1.y+v1.w*line.p1.z;

 matinv(&a[0][0]);
```

```
cut_point->x = a[0][0]*d + a[0][1]*d1 + a[0][2]*d2;
cut_point->y = a[1][0]*d + a[1][1]*d1 + a[1][2]*d2;
cut_point->z = a[2][0]*d + a[2][1]*d1 + a[2][2]*d2;
if (cosa0<= PLANETHICK) return(PARTCUT1);
if (cosa1<= PLANETHICK) return(PARTCUT0);

}

rebuild_planeq()
{
 struct count{
        short n
        };
struct count *normalcount;
float x,y,z,length;
int i,j,v1,v2,v3,vtmp,poly_index,jj,num_vert;
short *ptr;

        if (normalcount != NULL ) free(normalcount);
                normalcount=
                 (struct count *)malloc(free_vert * sizeof(struct count));
        for (i=0;i<free_vert;i++) {
                normalcount[i].n = 0;
                vertlst[i].normal.x = vertlst[i].normal.y
                = vertlst[i].normal.z =0.0;
                }

        /* rebuild planeq A,B,C,D and normal at ehch vertex          */
        for (i=0;i<nobj;i++){
        poly_index = objlst[i].index;
        for (j=0;j<objlst[i].npoly;j++){

                linking(&poly_index);

                ptr = polylst[poly_index].pvert.ptr;
                num_vert = polylst[poly_index].npvert;
                polylst[poly_index].planq.a = polylst[poly_index].planq.b
                = polylst[poly_index].planq.c = polylst[poly_index].planq.d = 0.0;
                v3 = *ptr++;
                v2 = *ptr++;
                v1 = *ptr;
                for (jj = 0; jj < 3; jj++)
                {
                polylst[poly_index].planq.a += vertlst[v1].vertex.y*
                                (vertlst[v2].vertex.z - vertlst[v3].vertex.z);
                polylst[poly_index].planq.b += vertlst[v1].vertex.x *
                                (vertlst[v3].vertex.z - vertlst[v2].vertex.z);
                polylst[poly_index].planq.c += vertlst[v1].vertex.x *
                                (vertlst[v2].vertex.y - vertlst[v3].vertex.y);
                polylst[poly_index].planq.d += vertlst[v1].vertex.x *
                                ((vertlst[v3].vertex.y * vertlst[v2].vertex.z) -
                                 (vertlst[v2].vertex.y * vertlst[v3].vertex.z));
                vtmp = v1; v1 = v2; v2 = v3; v3 = vtmp;
                }

                x = polylst[poly_index].planq.a;
                y = polylst[poly_index].planq.b;
```

```
                z = polylst[poly_index].planq.c;
                length = sqrt( x*x + y*y + z*z);
                ptr = polylst[poly_index].pvert.ptr;
                for (jj = 0; jj < polylst[poly_index].npvert;jj++)
                {
                vtmp = *ptr++;
                vertlst[vtmp].normal.x += x/length;
                vertlst[vtmp].normal.y += y/length;
                vertlst[vtmp].normal.z += z/length;
                normalcount[vtmp].n++;
                }
        poly_index++;
        } /* end of poly. j loop */
} /* end of obj. i loop        */




        for (i = 0; i < free_vert; i++)
                {
                if (normalcount[i].n != 0) {
                        vertlst[i].normal.x /= normalcount[i].n;
                        vertlst[i].normal.y /= normalcount[i].n;
                        vertlst[i].normal.z /= normalcount[i].n;
                        }
                }

/*  Free the temporary storage */
        free(normalcount);

} /* end of rebuild_planeq            */



new_cut_plane(cut_normal,plane_locate,new_plane_ptr,n_vert_ptr)
VECTOR cut_normal;
POINT plane_locate;
POINT *new_plane_ptr;
int *n_vert_ptr;
{
int i,j,k,n_vert,n_cut,search,plane_index,k1,flag;
int kk,jj;
POINT box[8],new_point;
LINE cut_line;
short temp_index0,temp_index1,e_index,pass;

struct {
        float x;
        float y;
        float z;
        short e_index;
        short v_index0;
        short v_index1;
        short p_index;
        short flag;
        }cut_point[12];
 VECTOR vtemp;
```

```
POINT new_plane[6];
float dot_product();
float cosa[4];


box[0].x = bbox[0][0]; box[0].y = bbox[1][0]; box[0].z = bbox[2][0];
box[1].x = bbox[0][1]; box[1].y = bbox[1][0]; box[1].z = bbox[2][0];
box[2].x = bbox[0][1]; box[2].y = bbox[1][0]; box[2].z = bbox[2][1];
box[3].x = bbox[0][0]; box[3].y = bbox[1][0]; box[3].z = bbox[2][1];

box[4].x = bbox[0][0]; box[4].y = bbox[1][1]; box[4].z = bbox[2][0];
box[5].x = bbox[0][1]; box[5].y = bbox[1][1]; box[5].z = bbox[2][0];
box[6].x = bbox[0][1]; box[6].y = bbox[1][1]; box[6].z = bbox[2][1];
box[7].x = bbox[0][0]; box[7].y = bbox[1][1]; box[7].z = bbox[2][1];

n_cut = 0;
printf ("\n%3.3f %3.3f %3.3f",plane_locate.x,plane_locate.y,plane_locate.z);

for (i=0;i<6;i++){
        for (j=0;j<4;j++){
                k = vertex_index[i][j];
                vtemp.u = box[--k].x - plane_locate.x;
                vtemp.v = box[k].y - plane_locate.y;
                vtemp.w = box[k].z - plane_locate.z;
                cosa[j] = dot_product(&cut_normal,&vtemp);
                }
        if (cosa[0] <=0.0 && cosa[1] <=0.0 && cosa[2] <=0.0 && cosa[3] <=0.0 )
        goto jump;



        for (j=0;j<4;j++){
        k = vertex_index[i][j];
        k1 = ( j == 3 ) ? vertex_index[i][0] : vertex_index[i][j+1];
        cut_line.p0.x = box[--k].x;
        cut_line.p0.y = box[k].y;
        cut_line.p0.z = box[k].z;
        cut_line.p1.x = box[--k1].x;
        cut_line.p1.y = box[k1].y;
        cut_line.p1.z = box[k1].z;




        flag = plane_cut_line(cut_normal,plane_locate,cut_line,&new_point);
        if (flag == PARTCUT1 || flag == ENDCUT1
        || flag == PARTCUT0 || flag == ENDCUT0 || flag == ONPLANE ){
                cut_point[n_cut].x = new_point.x;
                cut_point[n_cut].y = new_point.y;
                cut_point[n_cut].z = new_point.z;
                cut_point[n_cut].v_index0 = k;
                cut_point[n_cut].v_index1 = k1;
                cut_point[n_cut].flag = UNPICKED;
                if (flag == ENDCUT1 || flag == ENDCUT0  || flag == ONPLANE)
                        cut_point[n_cut].e_index = (flag == ENDCUT1) ? k1:k;
                else
                        cut_point[n_cut].e_index =  NOTEND;
```

```
                cut_point[n_cut++].p_index = i;
                if (flag == ONPLANE ){
                cut_point[n_cut].x = new_point.x;
                cut_point[n_cut].y = new_point.y;
                cut_point[n_cut].z = new_point.z;
                cut_point[n_cut].v_index0 = k;
                cut_point[n_cut].v_index1 = k1;
                cut_point[n_cut].flag = UNPICKED;
                cut_point[n_cut].e_index =  k1;
                cut_point[n_cut++].p_index = i;
                }
                if (n_cut>12){
                        printf("n_cut more than 12 !!");
                        shut_down_core1();
                        exit(1);
                        }
                }
        } /* end of j loop (line loop)          */
        jump: j=0;
}/* end of i loop (plane loop)        */


cut_point[0].flag = PICKED;
temp_index0 = cut_point[0].v_index0;
temp_index1 = cut_point[0].v_index1;
e_index = cut_point[0].e_index;
plane_index = cut_point[0].p_index;
new_plane[0].x = cut_point[0].x;
new_plane[0].y = cut_point[0].y;
new_plane[0].z = cut_point[0].z;

if (cut_point[0].e_index == NOTEND){
        for (i=1;i<n_cut;i++)
        if  ((temp_index0 == cut_point[i].v_index0
        && temp_index1 == cut_point[i].v_index1) ||
        (temp_index0 == cut_point[i].v_index1
        && temp_index1 == cut_point[i].v_index0)) cut_point[i].flag = END;
        }
else {
        for (i=1;i<n_cut;i++)
        if (e_index == cut_point[i].e_index
        && plane_index != cut_point[i].p_index) cut_point[i].flag = END;
        };


n_vert = 1;
pass =0;
search = BEGIN;
        kk = 1;
while (search == BEGIN ){
        if (e_index != NOTEND  ) {
                for (i=1;i<n_cut;i++)
                        if (plane_index == cut_point[i].p_index
                        && e_index== cut_point[i].e_index)
                                        cut_point[i].flag = PICKED;

                        for (i=1;i<n_cut;i++)
```

```
                        if (plane_index == cut_point[i].p_index
                        && cut_point[i].flag == UNPICKED ){
                                cut_point[i].flag = PICKED;
                                if (e_index != cut_point[i].e_index){
                                temp_index0 = cut_point[i].v_index0;
                                temp_index1 = cut_point[i].v_index1;
                                e_index = cut_point[i].e_index;
                                new_plane[n_vert].x = cut_point[i].x;
                                new_plane[n_vert].y = cut_point[i].y;
                                new_plane[n_vert++].z = cut_point[i].z;
                                }
                        }
                }
        else {
        i=1;
        while(i<n_cut){
        if (plane_index == cut_point[i].p_index &&
                cut_point[i].flag == UNPICKED ){
                cut_point[i].flag = PICKED;
                temp_index0 = cut_point[i].v_index0;
                temp_index1 = cut_point[i].v_index1;
                e_index = cut_point[i].e_index;
                new_plane[n_vert].x = cut_point[i].x;
                new_plane[n_vert].y = cut_point[i].y;
                new_plane[n_vert++].z = cut_point[i].z;
                break;
                }
        i++;
        }
                }
        jj=1;
while(jj<n_cut){
        if (e_index == NOTEND ){
                if (cut_point[jj].flag != PICKED
                &&((temp_index0 == cut_point[jj].v_index0
                && temp_index1 == cut_point[jj].v_index1)
                ||(temp_index0 == cut_point[jj].v_index1
                && temp_index1 == cut_point[jj].v_index0))){
                        cut_point[jj].flag = PICKED;
                        plane_index = cut_point[jj].p_index;
                        break;
                        }
                if (cut_point[jj].flag == END
                        &&((temp_index0 == cut_point[jj].v_index0
                        && temp_index1 == cut_point[jj].v_index1)
                        ||(temp_index0 == cut_point[jj].v_index1
                        && temp_index1 == cut_point[jj].v_index0))){
                        search = END;
                        break;
                        }
                }
         if (e_index != NOTEND ){
                if (e_index == cut_point[jj].e_index
                        && cut_point[jj].flag == UNPICKED){
                        cut_point[jj].flag = PICKED;
                        plane_index = cut_point[jj].p_index;
                        break;
                        }
```

```
                        if (e_index == cut_point[jj].e_index
                                && cut_point[jj].flag == END){
                                search = END;
                                break;
                                }
                        }
                jj++;
        }/* end of jj       loop    */
        if ( pass++ > 6 ) break;

}/* end of search loop */

for (i=0;i<n_vert;i++){
        new_plane_ptr->x = new_plane[i].x;
        new_plane_ptr->y = new_plane[i].y;
        new_plane_ptr->z = new_plane[i].z;
        new_plane_ptr++;
        }

*n_vert_ptr = n_vert;

}/* end of new_cut_plane          */



#include <usercore.h>
#include <sun/fbio.h>
#include <stdio.h>
#include <math.h>
#include "model.h"


extern  POINT cut_plane[];
extern  int     nvert,npoly,face_remove,num_shade_level;
extern  int     npvert[],*pvertptr[],pvert[],indxlist[],number;
extern  float   planeq[][4],vertices[][3],normal[][3];
extern  short   mycolor1,cindex[],plan_info[],cline[];
extern  float   xmax,xmin,ymax,ymin,zmax,zmin,xcent
                        ,ycent,zcent,length,emin,emax;
extern  float   dxlist[],dylist[],dzlist[]
                        ,xlist[],ylist[],zlist[],n_vert[][3],line_vert[][3];


/* line variables  */
extern   int nline,nlvert,line_index_list[],npline[];

/* Externals */
extern  nobj,npoly,nvert;
extern  free_vert,free_poly,free_obj,free_pvert;
extern  old_vert,old_poly,old_pvert;
extern  float bbox[][2];

extern struct image_status_format image_status;
extern struct obj *objlst;
extern struct poly *polylst;
extern struct vert *vertlst;
extern struct pvert *pvertlst;
```

```
defin_color()
{
int     i;
float   x,y,z,x0,y0,z0,length;

        map_ndc_to_world_3(-348.,348.,-870.,&x,&y,&z);
        map_ndc_to_world_3(0.0,0.0,0.0,&x0,&y0,&z0);
        x -=x0; y -=y0; z -=z0;
        length = sqrt(x*x +y*y + z*z);
        if (length != 0.0)
                {
                x /=length;y /=length;z /=length;
                }

        for (i=0; i< nvert; i++)
                {
                cindex[i]=
                fabs(normal[i][0]*x+normal[i][1]*y+normal[i][2]*z) * 255.0;
                if (cindex[i]<1.0) cindex[i]=1.0;
                if (cindex[i]>255.0 ) cindex[i]=255.0;
                }


}       /* end of defin_color      */

backsurface_check()
{
int i,j,k,p;
int *ptr;
float x[6],y[6],z[6],nx,ny,nz,c,ai,aj,bi,bj,ci,cj,di,dj;;
for (p=0;p<npoly;p++)
        {
        ptr=pvertptr[p];k=npvert[p];
        for (i=0;i<k;i++)
        {
        j= *ptr++;
        xlist[i]=vertices[j][0];
        ylist[i]=vertices[j][1];
        zlist[i]=vertices[j][2];
        }
        for (i=0;i<npvert[p];i++)
                map_world_to_ndc_3(xlist[i],ylist[i],zlist[i],&x[i],&y[i],&z[i]);

        c=0.0;
        ai=x[1]-x[0];aj=y[1]-y[0];
        bi=x[2]-x[1];bj=y[2]-y[1];
        ci=x[3]-x[2];cj=y[3]-y[2];
        di=x[0]-x[3];dj=y[0]-y[3];

        c=ai*bj-aj*bi+bi*cj-bj*ci+ci*dj-cj*di+di*aj-dj*ai;

        if (c <0.0 && plan_info[p]<0 || c >0.0 && plan_info[p]>0 )
                  plan_info[p]= -plan_info[p];
        }

}       /* end of backsurface_check        */
```

```
ndc_to_world()
{
int i,j,k;
float x,y,z;
for (i=0;i<nvert;i++)
        {
        n_vert[i][0] *=1020;
        n_vert[i][1] *=1020;
        n_vert[i][2] *=1020;
        if (i==0)
                {
                xmin=xmax=n_vert[i][0];
                ymin=ymax=n_vert[i][1];
                zmin=zmax=n_vert[i][2];
                }
        else
                {
                xmin= (xmin>n_vert[i][0])? n_vert[i][0]:xmin;
                xmax= (xmax<n_vert[i][0])? n_vert[i][0]:xmax;
                ymin= (ymin>n_vert[i][1])? n_vert[i][1]:ymin;
                ymax= (ymax<n_vert[i][1])? n_vert[i][1]:ymax;
                zmin= (zmin>n_vert[i][2])? n_vert[i][2]:zmin;
                zmax= (zmax<n_vert[i][2])? n_vert[i][2]:zmax;
                emin= (emin> cindex[i]) ? cindex[i]:emin;
                emax= (emax< cindex[i]) ? cindex[i]:emax;

                }
        }
xcent=xmin+(xmax-xmin)/2;
ycent=ymin+(ymax-ymin)/2;
zcent=zmin+(zmax-zmin)/2;
for (i=0;i<nvert;i++)
        {
        if (cindex[i]>1)
                cindex[i]=(int) (cindex[i]-emin)*num_shade_level/(emax-emin);
        }
}       /* end of ndc_to_world   */




drawline()
{
        int i,j,k,np,color,v_index;
        int *ptr;

        ptr= &line_index_list[0];
        for (i=0;i<nline;i++)
        {
        np=npline[i];
        color = cline[i];
        if (color < 0)
                ptr +=np;
        else
```

```
                {
                set_linewidth(0.3);
                set_line_index(color);
                for (j=0;j<np;j++)
                        {
                        v_index = *ptr;
                        xlist[j] = line_vert[v_index][0];
                        ylist[j] = line_vert[v_index][1];
                        zlist[j] = line_vert[v_index][2];
                        ptr++;
                        }
                        polyline_abs_3(xlist,ylist,zlist,np);
                }
        }
        set_linewidth(0.0);
}       /* end of drawline          */

draw_axis()
{
int color;
/*
set_line_index(100);
move_abs_3(0.0,0.0,0.0);
line_rel_3(200.0,0.0,0.0);
set_line_index(60);
move_abs_3(0.0,0.0,0.0);
line_rel_3(0.0,200.0,0.0);
set_line_index(80);
move_abs_3(0.0,0.0,0.0);
line_rel_3(0.0,0.0,200.0);
set_line_index(4);
*/
} /* end of draw axis        */


define_color()
{
int     i;
float x,y,z,x0,y0,z0,length;

map_ndc_to_world_3(-348.,348.-870.,&x,&y,&z);
map_ndc_to_world_3(0.0,0.0,0.0,&x0,&y0,&z0);
x -=x0; y -=y0; z -=z0;
length=sqrt(x*x+y*y+z*z);
if (length !=0.0)
        {
        x /=length;y /=length; z /=length;
        }

for (i=0;i<nvert;i++)
        {
        cindex[i]=(int)fabs(normal[i][0]*x+normal[i][1]*y+normal[i][2]*z)* 254.0;
        if (cindex[i]<1.0) cindex[i]=1.0;
        if (cindex[i]>255.0) cindex[i]=255.0;
        }

}       /* end of define_color      */
```

```
world_to_ndc()
{
int i,j,k;
for (i=0;i<nvert;i++)
        {
map_world_to_ndc_3(vertices[i][0],vertices[i][1],vertices[i][2],
        &n_vert[i][0],&n_vert[i][1],&n_vert[i][2]);
        }
}       /* end world_to _ndc     */



draw_box()
{
int i;
float x[5],y[5],z[5];
set_linewidth(0.3);
set_line_index(BOX_COLOR);
x[0]=x[4]=x[3]=bbox[0][0];
x[1]=x[2]=bbox[0][1];
z[0]=z[1]=z[4]=bbox[2][0];
z[2]=z[3]=bbox[2][1];
y[0]=y[1]=y[2]=y[3]=y[4]=bbox[1][0];
polyline_abs_3(x,y,z,5);
y[0]=y[1]=y[2]=y[3]=y[4]=bbox[1][1];
polyline_abs_3(x,y,z,5);

y[0]=y[4]=y[3]=bbox[1][0];
y[1]=y[2]=bbox[1][1];
z[0]=z[1]=z[4]=bbox[2][0];
z[2]=z[3]=bbox[2][1];
x[0]=x[1]=x[2]=x[3]=x[4]=bbox[0][1];
polyline_abs_3(x,y,z,5);
x[0]=x[1]=x[2]=x[3]=x[4]=bbox[0][1];
polyline_abs_3(x,y,z,5);


x[0]=x[4]=x[3]=bbox[0][0];
x[1]=x[2]=bbox[0][1];
y[0]=y[1]=y[4]=bbox[1][0];
y[2]=y[3]=bbox[1][0];
y[2]=y[3]=bbox[1][1];
z[0]=z[1]=z[2]=z[3]=z[4]=bbox[2][0];
polyline_abs_3(x,y,z,5);
z[0]=z[1]=z[2]=z[3]=z[4]=bbox[2][1];
polyline_abs_3(x,y,z,5);
draw_axis();

}       /* end of draw_box       */

drawface(p)
int p;
{

int i,j,k;
int *ptr;
float x[6],y[6],z[6],nx,ny,nz,c,ai,aj,bi,bj,ci,cj,di,dj;;
```

```
ptr=pvertptr[p];
for (i=0;i<npvert[p];i++)
        {
        j= *ptr++;
        xlist[i]=vertices[j][0];
        ylist[i]=vertices[j][1];
        zlist[i]=vertices[j][2];
        }
for (i=0;i<npvert[p];i++)
                map_world_to_ndc_3(xlist[i],ylist[i],zlist[i],&x[i],&y[i],&z[i]);

        c=0.0;
        ai=x[1]-x[0];aj=y[1]-y[0];
        bi=x[2]-x[1];bj=y[2]-y[1];
        ci=x[3]-x[2];cj=y[3]-y[2];
        di=x[0]-x[3];dj=y[0]-y[3];

        c=ai*bj-aj*bi+bi*cj-bj*ci+ci*dj-cj*di+di*aj-dj*ai;

xlist[4]=xlist[0];ylist[4]=ylist[0];zlist[4]=zlist[0];
if (c <0.0 )
                polyline_abs_3(xlist,ylist,zlist,npvert[p]+1);


}       /* end of drawface      */



drawobj()
{
int i;
float x[5],y[5],z[5];
set_line_index(LINE_COLOR);
if (image_status.special == TRUE)
        {for (i=618;i<npoly;i++)
                drawface(i);
        }
else
        for (i=0;i<npoly;i++)
                drawface(i);

draw_axis();

} /* end of drawobj       */

draw_wireframe()
{
short *ptr;
struct vert *vertlstptr;

int i,j,k,index,num_vert,poly_index;


set_line_index(LINE_COLOR);
for (i=0;i<nobj;i++){
        poly_index = objlst[i].index;
```

```
        for (j=0;j<objlst[i].npoly;j++){
                linking(&poly_index);
                ptr = polylst[poly_index].pvert.ptr;
                num_vert = polylst[poly_index++].npvert;
                for (k=0;k<num_vert;k++){
                                vertlstptr = vertlst + *ptr++;
                                xlist[k] = vertlstptr->vertex.x;
                                ylist[k] = vertlstptr->vertex.y;
                                zlist[k] = vertlstptr->vertex.z;
                                }
                xlist[k] = xlist[0];ylist[k] = ylist[0]; zlist[k] = zlist[0];
                polyline_abs_3(xlist,ylist,zlist,num_vert+1);
                }/* end of j poly loop      */
        }/* end of i obj loop          */
} /* end of draw_wirefram          */

drawobj1()
{
int i,j,k,index,num_vert,poly_index;
float x,y,z,x0,y0,z0,length;
short *ptr,*colorlst,color_level;
struct vert *vertlstptr;

        map_ndc_to_world_3(-348.0,348.0,-870.0,&x,&y,&z);
        map_ndc_to_world_3(0.0,0.0,0.0,&x0,&y0,&z0);
        x -= x0;y -= y0; z -= z0;
        length = sqrt(x*x+y*y+z*z);
        if (length != 0.0)
                {
                x /=length;y /=length; z /=length;
                }

        colorlst = (short *)malloc(free_vert * sizeof(short));
        for (i=0;i<free_vert;i++){
                colorlst[i]= fabs
                        (vertlst[i].normal.x*x+vertlst[i].normal.y*y
                        +vertlst[i].normal.z*z)*1000.;
                if (i==0)
                        emin=emax=colorlst[0];
                else
                        {
                        emin= (emin> colorlst[i]) ? colorlst[i]:emin;
                        emax= (emax< colorlst[i]) ? colorlst[i]:emax;
                        }
                }

         for (i=0;i<free_vert;i++)
                        colorlst[i] =  (colorlst[i]>1) ?
                        ((int)(colorlst[i]-emin)*num_shade_level/(emax-emin)):1;


         for (i=0;i<nobj;i++){
                poly_index = objlst[i].index;

                for (j=0;j<objlst[i].npoly;j++){
                /*
                again3:  while (polylst[poly_index].info == DELETE) poly_index++;
```

```
        if (polylst[poly_index].info < 0 )
                        poly_index = polylst[poly_index].pvert.index;
        if (polylst[poly_index].info == DELETE) goto again3;
                */
                        linking(&poly_index);
                        ptr = polylst[poly_index].pvert.ptr;
                        num_vert = polylst[poly_index].npvert;

                        color_level = (polylst[poly_index++].info-1)*num_shade_level+1;
                        for (k=0;k<num_vert;k++){
                                        indxlist[k]=colorlst[*ptr]+color_level;
                                        vertlstptr = vertlst + *ptr++;
                                        xlist[k] = vertlstptr->vertex.x;
                                        ylist[k] = vertlstptr->vertex.y;
                                        zlist[k] = vertlstptr->vertex.z;
                                }
                                set_vertex_indices(indxlist,num_vert);
                                polygon_abs_3(xlist,ylist,zlist,num_vert);
                        }/* end of j poly loop       */
        }/* end of i obj loop        */
        free(colorlst);

} /*     end of drawobj1          */


image_switch()
{
if (face_remove)
        {
        new_frame();
        face_remove=FALSE;
        }
 else
        {
        if (image_status.wire_frame)
                {
                delete_retained_segment(WIREFRAME);
                create_retained_segment(WIREFRAME);
                draw_wireframe();
                close_retained_segment(WIREFRAME);
                }
         if (image_status.out_box)
                {
                delete_retained_segment(OUTBOX);
                create_retained_segment(OUTBOX);
                 draw_box();
                close_retained_segment(OUTBOX);
                }
        if (image_status.line)
                {
                delete_retained_segment(DRAWLINE);
                create_retained_segment(DRAWLINE);
                drawline();
                close_retained_segment(DRAWLINE);
                }
        }

}
```

```
draw_cut_plane(n_point) int n_point;
{int i;
move_abs_3(cut_plane[0].x,cut_plane[0].y,cut_plane[0].z);
for (i=0;i<n_point;i++){
                    xlist[i] = cut_plane[i].x;
                    ylist[i] = cut_plane[i].y;
                    zlist[i] = cut_plane[i].z;
                    }
xlist[i] = cut_plane[0].x;ylist[i] = cut_plane[0].y;zlist[i] = cut_plane[0].z;
set_line_index(MENU_TEXT_COLOR);
polyline_abs_3(xlist,ylist,zlist,n_point+1);
} /* end of draw_cut_plane          */



#include <math.h>
#include"model.h"

/* Cross_product,dot_product and vector normalization      */


/* ************ Dot_product ******************/
/* Input the points of vector A and B          */
/* Return value : cos(theta)                              */


float dot_product(pva,pvb)          VECTOR *pva,*pvb;
{
float cos_theta;
          normalize(pva);
          normalize(pvb);
          cos_theta=((pva->u)*(pvb->u) + (pva->v)*(pvb->v) + (pva->w)*(pvb->w));
          return(cos_theta);

}         /* end of dot_product      */

/*************** Normalize*****************/
/* Input the point of the vector which will be normalized      */

normalize(pv)    VECTOR *pv;
{
float m;
          m= sqrt((pv->u)*(pv->u) + (pv->v)*(pv->v) + (pv->w)*(pv->w));
          pv->u / =m;
          pv->v /=m;
          pv->w /=m;
} /* end of vector_normalize          */

/************ Cross_product **************/
cross_product(pvc,pva,pvb)
          VECTOR  *pva,*pvb,*pvc;
{
          /*normalize(pva);
          normalize(pvb);*/
```

```
        pvc->u = (pva->v) * (pvb->w) - (pva->w) * (pvb->v);
        pvc->v = (pva->w) * (pvb->u) - (pva->u) * (pvb->w);
        pvc->w = (pva->u) * (pvb->v) - (pva->v) * (pvb->u);
/*
        if((pvc->u==0) && (pvc->v==0) && (pvc->w==0))
        {
                if (pvb->u==0) {pvc->u = 1.0;  pvc->v=0.0;  pvc->w = 0.0;}
                else
                {
                        if ( pvb->v==0) { pvc->u = -1.0/pvb->w; pvc->v=0.0; pvc->w = 1.0/pvb->u;}
                else
                {
                        pvc->u = 1.0;
                        pvc->w = 1.0;
                        pvc->v = -( pvb->u + pvb->w) / pvb->v;
                }
                }
                normalize(pvc);
                }

*/
} /* end of cross_product */


matinv(ptr)
        float *ptr;

{
        short index[3][2], ipivot[3];
        float pivot[3],mtxout[3][3];
        short row, colum;
        float max;
        short i, j, k, l;


        for (i=0;i<3;i++)
                for (j=0;j<3;j++)
                        mtxout[i][j] = *(ptr+i*3+j);

        for (j = 0; j < 3; j++)
                ipivot[j] = 0;


        for (i = 0; i < 3; i++) {
                max = 0.0;
                /* Search for pivot element */
                for (j = 0; j < 3; j++) {
                        if ( ipivot[j] == 1)
                                continue;
                        for (k = 0; k < 3; k++) {
                                if (ipivot[k] == 1)
                                        continue;
                                if (ipivot[k] > 1)
                                        return( 0);
                                if (fabs(max) < fabs(mtxout[j][k])) {
                                        row = j;
                                        colum = k;
                                        max = mtxout[j][k];
```

```
                        }
                    }
                }

                /* Row intercahnge */
                ipivot[colum] += 1;
                if (row != colum) {
                    for (l = 0; l < 3; l++) {
                        max = mtxout[row][l];
                        mtxout[row][l] = mtxout[colum][l];
                        mtxout[colum][l] = max;
                    }
                }

                index[i][0] = row;
                index[i][1] = colum;
                pivot[i] = mtxout[colum][colum];

                mtxout[colum][colum] = 1.0;
                for (l = 0; l < 3; l++)
                    mtxout[colum][l] /= pivot[i];

                for (j =0; j < 3; j++)
                    if (j != colum) {
                        max = mtxout[j][colum];
                        mtxout[j][colum] = 0.0;
                        for (l=0; l<3; l++)
                            mtxout[j][l] -= mtxout[colum][l] * max;
                    }
            }

            for (i = 0; i < 3; i++) {
                l = 3 - 1 - i;
                if (index[l][0] != index[l][1]) {
                    row = index[l][0];
                    colum = index[l][1];
                    for (k=0; k<3; k++) {
                        max = mtxout[k][row];
                        mtxout[k][row] = mtxout[k][colum];
                        mtxout[k][colum] = max;
                    }
                }
            }
            for (i=0;i<3;i++)
                for (j=0;j<3;j++)
                    *(ptr+i*3+j) = mtxout[i][j];
}




/*
* Module: load_file()
* Functions:
        1. Check data format;
        2. Allocate memory;
        3. Setup scaling parameters;
```

```
        4. Genarate object list;
        5. Compute the parameters of each poly.
        6. Compute the average normal at each vertex;
        7. Free the temporary storage.

* Input:
        data file: objects data, polygons data.
* Output:
        object list, polygons list, vertices list;
* Last change 10/8/88
*/


#include <math.h>
#include <stdio.h>
#include "model.h"

extern  int     npoly,face_remove,num_shade_level;
extern  int     npvert[],*pvertptr[],pvert[],indxlist[],number;
extern  float   bbox[][2],planeq[][4],vertices[][3],normal[][3];
extern  short   mycolor1,cindex[],plan_info[],cline[];
extern  float   xmax,xmin,ymax,ymin,zmax,zmin,xcent
                        ,ycent,zcent,length,emin,emax,scale;
extern  float   dxlist[],dylist[],dzlist[]
                        ,xlist[],ylist[],zlist[],n_vert[][3],line_vert[][3];


/* line variables  */
extern  int  nline,nlvert,line_index_list[],npline[];


extern  struct  image_status_format image_status;



/* Externals */
extern  nobj,npoly,nvert;
extern  free_vert,free_poly,free_obj,free_pvert;
extern  old_vert,old_poly,old_pvert;
extern  float bbox[3][2];

/* Externals */

extern struct obj *objlst;
extern struct poly *polylst;
extern struct vert *vertlst;
extern struct pvert *pvertlst;

load_file(filename)
char *filename;
{
int i,j;
int vtmp,v1,v2,v3;
float ftmp,maxd,offset[3];
float x,y,z,x0,y0,length;
FILE *fptr;
struct count{
        short n
```

```
        };
struct count *normalcount;

/* 1. Check data format; */
if ((fptr = fopen(filename, "r")) == NULL) {
                printf("Can't open file: %s\n", filename);
                return(1);
                }
fscanf(fptr, "%d%d%d", &nobj,&npoly,&nvert);
        if (nobj > MAXOBJ ) {
                printf("Too many objects\n");
                return(1);
                }
        if (npoly > MAXPOLY ) {
                printf("Too many polygons\n");
                return(1);
                }
        if (nvert > MAXVERT ) {
                printf("Too many verices\n");
                return(1);
                }
        if (nobj<=0 || npoly <=0 || nvert <=2 ){
                printf("Check your data! \n");
                return(1);
                }

/*  2. Allocate memory */
if (objlst != NULL ) free(objlst);
objlst = (struct obj *)malloc(nobj * sizeof(struct obj));

if (polylst != NULL ) free(polylst);
polylst = (struct poly *)malloc((npoly+MAXNEWPOLY) * sizeof(struct poly));

if (vertlst != NULL ) free(vertlst);
vertlst = (struct vert *)malloc((nvert+MAXNEWVERT) * sizeof(struct vert));

if (pvertlst != NULL ) free(pvertlst);
pvertlst =(struct pvert *) malloc((MAXPVERT + MAXNEWPVERT) * sizeof(short));

/*  3. Setup scaling parameters      */
fscanf(fptr, "%f%f%f%f%f%f", &bbox[0][0], &bbox[0][1], &bbox[1][0],&bbox[1][1],
        &bbox[2][0], &bbox[2][1]);

maxd = 0.0;
        for (i = 0; i < 3; i++) {
                offset[i] = (bbox[i][0] + bbox[i][1]) / 2.0;
                bbox[i][0] -= offset[i];
                bbox[i][1] -= offset[i];
                if (bbox[i][0] > bbox[i][1]) {
                        ftmp = bbox[i][0];
                        bbox[i][0] = bbox[i][1];
                        bbox[i][1] = ftmp;
                        }
                if (maxd < bbox[i][1])
                        maxd = bbox[i][1];
                }
        scale = 1000.0 / maxd;
        for (i = 0; i < 3; i++) {
```

```
                bbox[i][0] *= scale;
                bbox[i][1] *= scale;
                }

free_vert=free_pvert=free_poly=free_pvert=0;

/* 4. Genarate object list */
        if (normalcount != NULL ) free(normalcount);
                normalcount= (struct count *)malloc((nvert )* sizeof(struct count));
        if (normalcount == NULL ) {
        printf("\nMerory too small!!!\n");
        exit(1);
        }
for (free_obj=0;free_obj<nobj;free_obj++)  /* objects input   */
        {
        fscanf(fptr, "%d%d",&objlst[free_obj].npoly, &objlst[free_obj].nvert);
        objlst[free_obj].index= free_poly;
/*      if (normalcount != NULL ) free(normalcount);
        normalcount= (struct count *)malloc(objlst[free_obj].nvert
                * sizeof(struct count));*/

        /* vertices input  */

        for (i = 0; i <objlst[free_obj].nvert; i++)
        {
        fscanf(fptr, "%f%f%f", &x, &y,&z);
        vertlst[free_vert].vertex.x = (x-offset[0])*scale;
        vertlst[free_vert].vertex.y = (y-offset[1])*scale;
        vertlst[free_vert].vertex.z = (z-offset[2])*scale;
        vertlst[free_vert].normal.x = vertlst[free_vert].normal.y
                                        = vertlst[free_vert].normal.z =0.0;
        normalcount[free_vert++].n = 0;
        }/*  end of vertices input   */

        /* polygons input          */
        for (i = 0; i <objlst[free_obj].npoly ; i++)
        {
        fscanf(fptr, "%d",&vtmp );
        polylst[free_poly].npvert = vtmp;
        if ((old_pvert + polylst[free_poly].npvert) > MAXPVERT)
                {
                printf("Check your data !! RUN OVER PVERTLIST.\n");
                return(3);
                }
        fscanf(fptr, "%d", &vtmp);
        polylst[free_poly].info=vtmp;
        polylst[free_poly].pvert.ptr = &pvertlst[free_pvert].index;
        for (j = 0; j <polylst[free_poly].npvert ; j++)
                {
                fscanf(fptr, "%d", &vtmp);
                pvertlst[free_pvert++].index = vtmp - 1;
                }
        polylst[free_poly].planq.a = polylst[free_poly].planq.b
        = polylst[free_poly].planq.c = polylst[free_poly].planq.d = 0.0;

                v1 = pvertlst[free_pvert-1].index;
                v2 = pvertlst[free_pvert-2].index;
                v3 = pvertlst[free_pvert-3].index;
```


C-3

```
/*      5. Compute the parameters A,B,C and D of each poly.          */

        for (j = 0; j < 3; j++)
                {
                polylst[free_poly].planq.a += vertlst[v1].vertex.y*
                                (vertlst[v2].vertex.z - vertlst[v3].vertex.z);
                polylst[free_poly].planq.b += vertlst[v1].vertex.x *
                                (vertlst[v3].vertex.z - vertlst[v2].vertex.z);
                polylst[free_poly].planq.c += vertlst[v1].vertex.x *
                                (vertlst[v2].vertex.y - vertlst[v3].vertex.y);
                polylst[free_poly].planq.d += vertlst[v1].vertex.x *
                                ((vertlst[v3].vertex.y * vertlst[v2].vertex.z) -
                                 (vertlst[v2].vertex.y * vertlst[v3].vertex.z));
                vtmp = v1; v1 = v2; v2 = v3; v3 = vtmp;
                }

/*      6. Compute the average normal at each vertex        */

                x = polylst[free_poly].planq.a;
                y = polylst[free_poly].planq.b;
                z = polylst[free_poly].planq.c;
                length = sqrt( x*x + y*y + z*z);

                for (j = 1; j <= polylst[free_poly].npvert;j++)
                {       /* accum normls */
                vtmp = pvertlst[free_pvert-j].index;
                vertlst[vtmp].normal.x += x/length;
                vertlst[vtmp].normal.y += y/length;
                vertlst[vtmp].normal.z += z/length;
                normalcount[vtmp].n++;
                }

        free_poly++;


        } /*     end of poly input           */
        j=0;
        for (i = old_vert; i < free_vert; i++)
                {
                vertlst[i].normal.x /= normalcount[j].n;
                vertlst[i].normal.y /= normalcount[j].n;
                vertlst[i].normal.z /= normalcount[j++].n;
                }

/* 7.  Free the temporary storage */


        old_poly=free_poly;
        old_pvert=free_pvert;
        old_vert=free_vert;

        } /* end of object input     */

/* 7.  Free the temporary storage */
        free(normalcount);

fclose(fptr);
```

```
return(NULL);

}       /* end of load_file()      */



#include <usercore.h>
#include <sun/fbio.h>
#include <stdio.h>
#include <math.h>

#include "model.h"

extern  int     n_button;
extern  float menu_x,menu_y,menu_h,menu_w,menu_f,window_fact;
extern  struct  menu_table_format menu_table[];
extern  struct  view_parameters_format          my_view_parameters;
extern  struct  image_status_format image_status;



int menu_select()
{
float mx,my,fx,fy,fz;
float x,y,z;
int i,j,k,button,mycase;
set_primitive_attributes(&PRIMATTS);
mycase = 0;
set_echo(LOCATOR, 1 ,1);
for (;;)
        {
        do
                await_any_button_get_locator_2(20000000,1,&button,&mx,&my);
        while(button == 0);
        if ((button==1)&&(mx>0.9)&&(my>0.01)&&(mx<0.98)&&(my<0.66))
                {
                mx=(mx-.9)*1000.0;
                my=my*767.0/0.75;
                for (k=0;k<n_button;k++)
                        if ((mx<menu_table[k].maxx)&&(mx>menu_table[k].minx)&&
                                (my<menu_table[k].maxy)&&(my>menu_table[k].miny))
                                 mycase=menu_table[k].button_id;

                break;
                }
        }
set_echo(LOCATOR,1,0);
return(mycase);

}       /* end of menu_select      */

int call_menu(menuname,flag)
        int menuname,flag;
{
```

```
int i;
 inquire_viewing_parameters(&my_view_parameters);
 set_menu_vw();
 set_segment_visibility(menuname,TRUE);
 i = menu_select();
 set_segment_visibility(menuname,flag);
 set_viewing_parameters(&my_view_parameters);
 return(i);
}        /* end of call menu        */


set_menu_vw()
{

        set_view_reference_point(0.0,0.0,0.0);

        set_view_plane_normal(0.0,0.0, -1.0);
        set_view_plane_distance(0.0);
        set_projection(PARALLEL,0.0,0.0,1.0);
        set_view_up_3(0.0,1.0,0.0);
        set_window(0.0,100.0,0.0,767.0);
        set_view_depth(0.0,1.0);
        set_window_clipping(FALSE);
        set_viewport_3(0.92,1.0,0.0,.75,0.0,1.0);
}       /* end of set_menu_view          */


init_menu()
{

int i,j,k;
        inquire_viewing_parameters(&my_view_parameters);
        set_menu_vw();
        create_retained_segment(MENU);
        set_segment_visibility(MENU,TRUE);

        build_menu_table();
                build_menu1();

/*
        build_menu();
        */
        close_retained_segment(MENU);

        set_viewing_parameters(&my_view_parameters);
}       /* end of init_menu        */

build_menu_table()
{
FILE *fptr;
short i,n,j;
int status;
char nam[16];
float px,py;

fptr = fopen("menu_file", "r");
fscanf(fptr, "%f%f%f%f%d%f", &menu_x,&menu_y,&menu_w,&menu_h,&n_button,&menu_f);
px= menu_x;
```

```
py= menu_y;

for (i=0;i<n_button;i++)
{
        for (j=0;j<16;j++) nam[j]=4;
        fscanf(fptr, "%d", &status);
        fscanf(fptr, "%d", &menu_table[i].button_id);
        fscanf(fptr, "%d", &menu_table[i].button_frame);

        j=0;
        do
                fscanf(fptr, "%c", &nam[j]);
                while (nam[j++]!=10);


        menu_table[i].button_status = status;

        for (j=0;j<16;j++)
                if ((nam[j] >=65 && nam[j] <=90) | | (nam[j] >=97 && nam[j] <=122))
                        menu_table[i].name[j]=nam[j];
                else
                        menu_table[i].name[j]=32;

        if (menu_table[i].button_status == TRUE)
                menu_table[i].maxx=px+menu_w-menu_h/menu_f;
        else
                menu_table[i].maxx=px+menu_w/2.0 -menu_h/menu_f;

        menu_table[i].minx=px+menu_h/menu_f;
        menu_table[i].maxy=py;
        menu_table[i].miny=py-menu_h;

        if (menu_table[i].button_status == TRUE)
                py -= menu_h;
        else
                {
                if ((px+menu_w/2.0)>= (menu_x+menu_w))
                        {
                        px = menu_x;
                        py -= menu_h;
                        }
                else
                        px +=menu_w/2.0;
                }

        }/* end of i loop   */
fclose(fptr);
} /* end of build_menu_table        */

build_menu1()
{
int i,j,k;
float px,py,factor,x[12],y[12],z[12];


set_linewidth(0.3);
set_line_index(MENU_BOX_COLOR);
```

```
factor=menu_h/menu_f;
px=menu_x;
py=menu_y;
for (i=0;i<n_button;i++)
        {
        x[0]=menu_table[i].minx-factor;
        y[0]=menu_table[i].miny+factor;
        x[1]=menu_table[i].minx;
        y[1]=menu_table[i].miny;
        x[2]=menu_table[i].maxx;
        y[2]=menu_table[i].miny;
        x[3]=x[2]+factor;
        y[3]=y[0];
        x[4]=x[3];
        y[4]=menu_table[i].maxy-factor;
        x[5]=menu_table[i].maxx;
        y[5]=menu_table[i].maxy;
        x[6]=x[1];
        y[6]=menu_table[i].maxy;
        x[7]=x[0];
        y[7]=y[4];
        x[8]=x[0];
        y[8]=y[0];
        move_abs_3(x[7],y[7],0.5);
        if (menu_table[i].button_frame == 0)
                {
                polyline_abs_2(x,y,9);
                move_abs_2(x[0],y[0]+factor*1.8);
                set_text_index(MENU_TEXT_COLOR);
                text(&menu_table[i].name[0]);
                }
         else
                {
                set_fill_index(menu_table[i].button_frame);
                polygon_abs_2(x,y,9);
                move_abs_2(x[0],y[0]+factor*1.6);
                set_text_index(129);
                text(&menu_table[i].name[0]);
                }

         if (menu_table[i].button_status== TRUE)
                {
                py -=menu_h;
                px = menu_x;
                }
          else
                {
                if ((px+menu_w/2.0)>= (menu_x+menu_w))
                        {
                        px = menu_x;
                        py -= menu_h;
                        }
                 else
                        px +=menu_w/2.0;
                }
        }
  set_linewidth(0.0);
  }       /* end of build_menu       */
```

```
#include <usercore.h>
#include <sun/fbio.h>
#include <stdio.h>
#include <math.h>

#include "model.h"

extern  float red[],grn[],blu[],dot,T,B,L,R,window_fact;
extern  struct   vwsurf  *our_surface;

shut_down_core1()
{
        terminate_device(KEYBOARD,1);
        deselect_view_surface(our_surface);
        terminate_view_surface(our_surface);
        terminate_core();

}       /* end of shut_down_core1        */


start_up_core()
{
int i;
FILE *fptr;

initialize_core(BASIC,SYNCHRONOUS,THREED);
our_surface->cmapsize = 256;
our_surface->cmapname[0]='\0';
if(initialize_view_surface(our_surface,TRUE)) exit(1);
initialize_device(BUTTON,1);
initialize_device(BUTTON,2);
initialize_device(BUTTON,3);
select_view_surface(our_surface);
set_light_direction(-0.45,0.45,-0.45);

set_shading_parameters(.5,.5,.5,0.5,7.,0,0);

initialize_device(KEYBOARD,1);
set_echo_surface(KEYBOARD,1,our_surface);
set_keyboard(1,80,"",1);
initialize_device(LOCATOR,1);
initialize_device( PICK, 1);
set_pick(1,0.001);

set_echo(LOCATOR,1,0);
set_echo_surface(LOCATOR,1,our_surface);
/*set_output_clipping(TRUE);*/
fptr = fopen("new_color.dat", "r");

inquire_color_indices(our_surface,0,255,red,grn,blu);
fscanf(fptr, "%f%f%f%f%f", &dot,&T,&B,&L,&R);
for (i=0;i<256;i++)
        fscanf(fptr, "%f", &red[i]);
for (i=0;i<256;i++)
        fscanf(fptr, "%f", &grn[i]);
```

```
        for (i=0;i<256;i++)
        fscanf(fptr, "%f", &blu[i]);
define_color_indices(our_surface,0,255,red,grn,blu);

fclose(fptr);

}       /* end of start_up_core   */


setvwpo(vx, vy, vz,bbox)
float vx, vy, vz,bbox[3][2];
        {
        int i;
        float diag, del, objdist, near;

        set_view_reference_point(vx, vy, vz);
        set_view_plane_normal(-vx, -vy, -vz);
        set_projection(PERSPECTIVE, 0., 0., 0.);
        set_view_plane_distance(256.0);
        if ((vx == 0.0) && (vz == 0.0))
                set_view_up_3(0.0, 0.0, vy);
        else
                set_view_up_3(0.0, 1.0, 0.0);
        set_window(-80.0*window_fact, 80.0*window_fact, -65.0*window_fact, 65.0*window_fact);
        diag = 0.0;
        for (i = 0; i < 3; i++) {
                del = bbox[i][1] - bbox[i][0];
                diag += del * del;
                }
        diag = sqrt(diag) / 2.0;
        objdist = sqrt( vx*vx + vy*vy + vz*vz);
        near = (diag >= objdist) ? objdist/2.0 : objdist-diag;
        set_view_depth( near, objdist + diag);
        set_window_clipping(TRUE);
        set_front_plane_clipping(TRUE);
        set_back_plane_clipping(TRUE);
        /*
        set_viewport_3(.17, .92, 0., .75, 0.0, 1.0);
        */
        }/* end of setvwpo          */

setvwpv()
{
        set_view_reference_point(0.0,0.0,0.0);
        set_view_plane_normal(0.0,0.0,-1.0);
        set_view_plane_distance(0.0);
        set_projection(PARALLEL,0.0,0.0,1.0);
        set_view_up_3(0.0,1.0,0.0);
        set_window(0.0,1023.0,0.0,767.0);
        set_view_depth(0.0,1.0);
        set_window_clipping(FALSE);
        set_viewport_3(0.0,1.0,0.0,.75,0.0,1.0);
}       /* end of setvwpv          */
```